IAS Guide to Writing a Device Handler Task

Order Number: AA–H278C–TC

This document contains instructions for writing a device handler task for a peripheral device that is not part of the standard hardware configuration.

Operating System Version: IAS Version 3.4

May 1990

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1990 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DDIF	IAS	VAX C
DEC	MASSBUS	VAXcluster
DEC/CMS	PDP	VAXstation
DEC/MMS	PDT	VMS
DECnet	RSTS	VR150/160
DECUS	RSX	VT
DECwindows	ULTRIX	
DECwrite	UNIBUS	
DIBOL	VAX	digital™

This document was prepared using VAX DOCUMENT, Version 1.2

Contents

FIGURES

TABLES

Preface

Purpose of the Manual

The intent of this manual is to provide the information necessary to enable system managers and programmers of IAS operating systems to create a device handler task for a peripheral I/O device that is not part of the standard system hardware configuration.

To use this manual efffectively, you should:

- Be familiar with the PDP-11 computer and its peripheral devices.
- Have a thorough understanding of the operation of the IAS system.
- Be able to write programs using the MACRO-11 assembly language.
- Be able to use the task builder program.

Structure of the Document

This manual consists of eight chapters and four appendixes.

- Chapter 1 introduces the general functional requirements for a device handler task.
- Chapter 2 describes the task's internal communication tables.
- Chapter 3 discusses the various functions that must be performed during task initialization.
- Chapter 4 describes the details of a task's main functions.
- Chapter 5 describes the interrupt service routine portion of a task.
- Chapter 6 contains generation and linking procedures necessary to incorporate the task into the host operating system.
- Chaper 7 discusses the system's error logging facility and how to interface the task to this facility.
- Chapter 8 discribes UMR handling for handler tasks that are to service DMA devices attached to the UNIBUS of a PDP-11/70 or an 11/44.
- Appendix A describes the IAS system subroutines that a task can use.
- Appendix B shows the format, content, and offsets for various system tables and lists.
- Appendix C shows the structure of the task's I/O status block.
- Appendix D directs the programmer to various device handler source programs which can be used as examples

Associated Documents

Documents that provide information related to the creation, installation, and use of device handler tasks are listed in the *IAS Master Index and Documentation Directory*.

1 Device Handler Task Functions

IAS provides a flexible, device-and-function-independent I/O capability that can support standard PDP-11 peripherals and special purpose devices. Peripheral device support is not an integral part of the Executive. It is provided by privileged tasks called device handlers.

I/O requests are issued by user tasks to logical units. The Executive maps the requests into physical device references using a set of device assignments. Each task has its own set of assignments that can be changed from the user's terminal or by the task during execution.

An I/O request is made by issuing a system directive, Queue I/O (QIO), to queue a request for a specified logical unit number (LUN). If the LUN is assigned to a physical unit and if the handler task that supports that unit is in memory and loaded or running, the request information is put into a request node buffer that is queued by priority in a request list for the specified physical unit.

The Executive does not attempt to interpret or validate the request; it only passes the request to a device handler task indicated by the LUN assignment. Interpretation and execution of the request are functions of the device handler task.

When an I/O request is issued by a user task, control is returned immediately to the task (contingent only upon the task priority). The user task has the option to suspend execution until I/O completion or to operate asynchronously.

The handler task can notify the user task of I/O completion by calling a system subroutine. You can use the subroutine to perform the notification in any of the following ways:

- Declaring a significant event and setting a specified event flag.
- Setting indicators in an I/O status block within the requesting task.
- Causing an asynchronous system trap for the requesting task.

Each I/O request contains an I/O function code that describes the operation to be performed. Device handler tasks must be able to interpret a set of standard I/O function codes in a manner that is appropriate to the indicated device.

Because most devices have device-dependent characteristics, it is not practical to implement all functions for all devices (for example, a read function is not implemented for a line printer). If an I/O request is issued that contains an illegal function code for a specified device, the handler task returns an error code indicating that the function is not implemented.

Device handler tasks also return a standard set of error conditions to the I/O status block. Appendix C describes the I/O status block.

The IAS system provides device handler tasks for standard Digital hardware (for example, DECtape, magnetic tape, line printers, A/D converters); however, when a nonstandard device is to be supported, the user must write the requisite device handler task. The user can write device handler tasks to control I/O for both single-unit and multiple-unit devices. Multiple-unit devices require more complex handlers than single-unit devices.

Device handler tasks contain five basic sections:

1 A table area to facilitate communication with system subroutines,

2 Initialization code to execute once the task is loaded,

3 Code to dequeue and service I/O requests,

4 An interrupt service routine (ISR), and

5 A power failure recovery section.

In some cases, for example pseudo device handlers, power failure recovery and/or an ISR is not required.

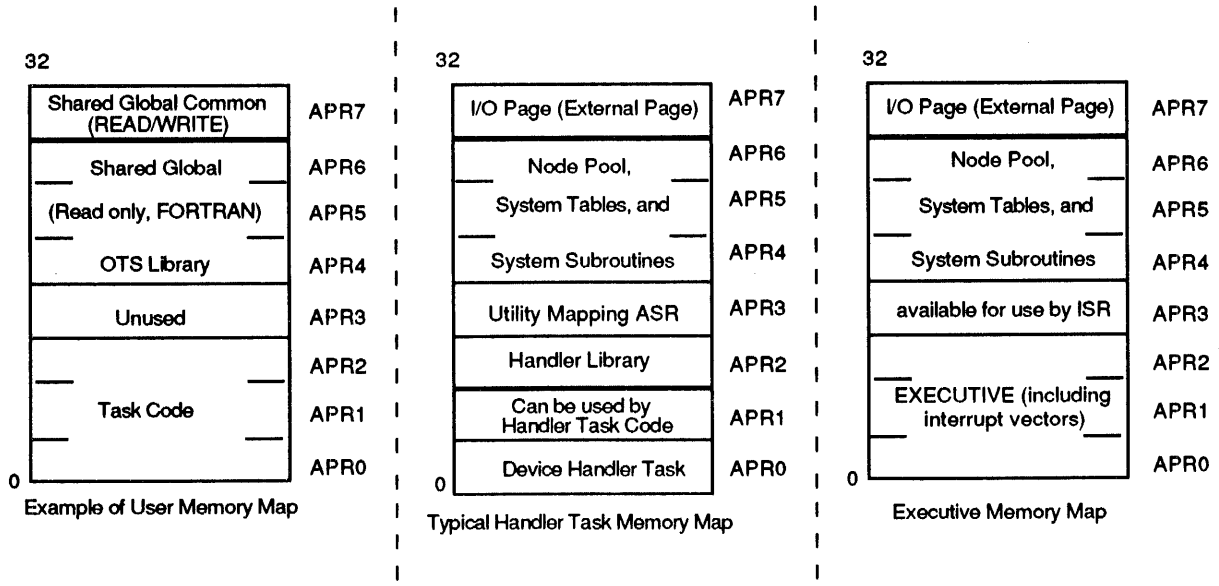
1.1 Executive Privileged Tasks

Because device handler tasks are executive privileged tasks, they have access to some Executive portions of memory. Portions of the handler task are commonly mapped with the Executive, thus enabling access to common areas of code. Figure 1–1 illustrates a typical memory map for a device handler task.

Note that APR3 is used by several Handler Library (HNDLIB) and System Communications area (SCOM) routines as a *scratch* mapping area. APR3 should not contain handler code. If the handler also uses APR3 as a scratch area, care should be taken that the data currently mapped by the APR is not required by any of the system routines. See also Section 6.2.

Handler tasks, like all other tasks, execute in User Mode, with the exception of the interrupt service routine, which executes in Kernel Mode.

Figure 1–1 Executive and Typical Handler Task Memory Maps



1.2 Queue I/O Directive

The Queue I/O (QIO) is the lowest level of task I/O. When a task issues a QIO directive, the directive parameter block (DPB) contains the information that the Executive requires to place the I/O request in the queue of the desired device handler. The DPB provides the following information:

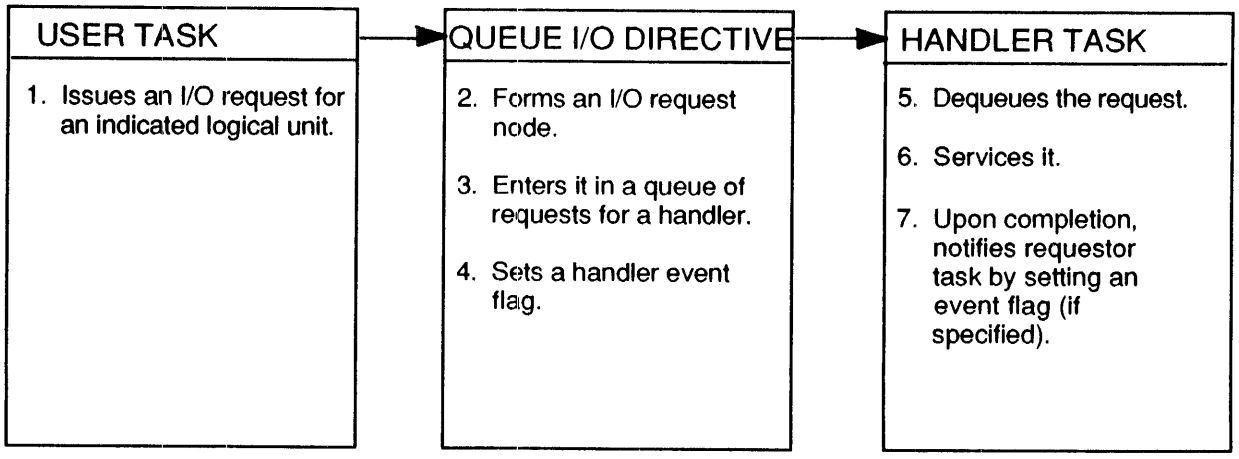
- An I/O function code,
- A logical unit number (LUN),
- An event flag number,
- A request priority,
- The address of the I/O status block, (optional),
- The address of the I/O done asynchronous trap (AST) entry point (optional), and
- A list of up to six parameters specific to the I/O function.

The *IAS System Directives Reference Manual* contains a detailed description of the QIO directive.

The Executive uses the LUN to determine the device handler task for which a request is intended, takes a node from the pool, and fills the node with I/O request parameters from the QIO DPB. A node containing I/O request parameters is called an I/O request node. The Executive places this node in the device handler task's queue. I/O requests are queued according to priority (normally that of the requesting task) so that higher priority requests receive faster service than lower priority requests. Requests of the same priority are queued in the order issued.

After queuing a request, the Executive sets an event flag to signal the handler task that an I/O request for the task has been queued. Figure 1–2 illustrates the flow of an I/O request.

Figure 1–2 Interface Between QIO and Device Handler Task



1.3 I/O Interrupts

When an I/O interrupt occurs, the handler task's interrupt service routine (ISR) must process the interrupt. Because the rapid processing of interrupts is essential, the Executive maps the ISR into Kernel address space 60000 through 77777 (APR3) and jumps (JMP) to it for execution whenever an interrupt occurs. Processing the interrupt in this fashion eliminates the need to switch from Kernel Mode to User Mode before processing the interrupt. It also eliminates the need to save and restore the user APRs. See Figure 1-1 for an illustration of the Executive memory map.

The initialization portion of the device handler task contains a call to a system subroutine (..CINT) to connect an interrupt service routine to a unique vector address. For multiple-unit handlers, this subroutine is called once for each vector. The subroutine creates a code sequence (contained in a node) that executes in Kernel Mode. The code sequence performs the following functions when an interrupt occurs:

1 Saves the current Kernel APR3 on the stack,

2 Sets Kernel APR3 to the address of the handler task's ISR,

3 Clears the corresponding bus activity bit in the I/O bit map used in error logging,

4 Sets specified condition codes for the unit in the Processor Status Word (condition code settings are frequently used to pass unit numbers when a common ISR is used for multiple units), and

5 Jumps to the ISR of the handler task.

1.4 System Subroutines

The IAS system provides subroutines that device handler tasks can call to perform standard functions. Some of the subroutines are contained in the system communications area (SCOM) and are available to any privileged task. Others are contained within the handler library routines (HNDLIB). Access the handler library routines either by mapping onto a shareable global area or extracting them from an object module library at task build time. Use of the Handler Library is described in Section 6.2 and Appendix A. Appendix A also describes the functions of the system subroutines.

System subroutine names have the format ..xxxx. The designation xxxx is a unique 4-character name (for example, ..CINT). Device handler tasks call the subroutines with the CALL macro, which generates a JSR to the subroutine using the program counter (JSR PC). The only exception is ..DISP, which is jumped to (JMP) rather than called.

Communicate with system subroutines through general registers and the program stack. An error return from a system subroutine causes the C condition code in the processor status word to be set. Routines such as ..IODN that do not detect errors can return with the C condition code either set or clear.

1.5 Processor Priorities

When a device handler task is running, it is often necessary to disable the host operating system's task switching function as well as the processor's interrupt facility while critical sections of handler code are being executed. Task switching is disabled by raising the processor priority level to 3. Interrupts and task switching are inhibited by raising the processor priority level to 7. The following macros are commonly used to disable and reenable task switching and interrupts. These macros are contained in the file RSXMAC.SML. The macros .INH0 and .ENB0 should be used as a matched pair, and so should .INH and .ENB.

Inhibit Task Switching:

```
                ;PROCESSOR STATUS WORD ADDRESS
.MACRO .INH0        ;INHIBIT TASK SWITCHING
MOV  PS.EXP,-(SP)   ;SAVE THE CURRENT PROGRAM
                ;STATUS
BIS  #140,PS.EXP    ;SET PRIORITY TO 3
.ENDM .INH0
```

Allow Task Switching:

```
                ;PROCESSOR STATUS WORD ADDRESS
.MACRO .ENB0        ;ALLOW TASK SWITCHING
CALL  ..ENB0        ;CALL THIS SYSTEM ROUTINE WITH
                ;THE PROGRAM STATUS ON THE
                ;TOP OF THE STACK
.ENDM .ENB0
```

Inhibit Interrupts and Task Switching:

```
                ;PROCESSOR STATUS WORD ADDRESS
.MACRO .INH         ;PREVENT INTERRUPTS
MOV  PS.EXP,-(SP)   ;SAVE THE CURRENT PROGRAM
                ;STATUS
BIS  #340,PS.EXP    ;SET PROCESSOR PRIORITY TO 7
.ENDM .INH
```

Reenable Interrupts and Task Switching:

```
                ;PROCESSOR STATUS WORD ADDRESS
.MACRO .ENB         ;ENABLE INTERRUPTS
MOV  (SP)+,PS.EXP   ;RESET PROGRAM STATUS TO
                ;PREVIOUS PRIORITY LEVEL
.ENDM .ENB
```

1.6 System Data Structures

A device handler task makes use of a number of data structures that are internal to the host operating system. These structures reside in SCOM, which is accessible only to privileged tasks (see Chapter 6). Detailed information describing these data structures is provided in Appendix B.

1.6.1 Physical Unit Directory (PUD)

The physical unit directory (PUD) is a system table that contains descriptive information for each physical device in the system. The table is created during system generation. Each PUD entry is 26 (decimal) words long. References to physical devices from other system tables point to the corresponding PUD entry.

1.6.2 System Task Directory (STD)

The system task directory (STD) is a table that provides information about each task installed in the system. The information recorded in a task's STD entry includes:

- Information required when the task is not active (for example, receive linked list listhead),
- Information required to load a task into memory (for example, task name, disk address of image)

1.6.3 Active Task List (ATL)

The active task list (ATL) contains an entry for each active task. Information contained in each entry includes event flag settings and I/O counts.

1.6.4 I/O Request Node

For each I/O request issued, the host operating system creates an I/O request node to record information about the request. This node is linked into the I/O Request Queue (IRQ) for the particular device handler to perform the I/O operation. The handler dequeues an I/O request node each time it is ready to perform another I/O operation.

The IRQ for the handler is a deque of I/O request nodes that are linked to the handler task's header, at offset H.IOQ. The *IAS Executive Facilities Reference Manual* describes the IRQ and the task header more fully.

1.7 Multiuser Device Handlers

Some tasks are built such that several copies of those tasks can run simultaneously. Such tasks are called multiuser tasks.

For some devices (for example, line printers, card readers) there is one hardware controller for each device unit. In such a case, it is convenient to write the device handler for just one unit, to build it as a multi-user task, and to run a separate copy of that task for each unit to be serviced. The handler does not have to cope with several simultaneous transfers, and each unit can operate at full speed.

The concept can be extended to systems that have several controllers of one kind where each controller governs several units. The handler can be written for one controller, then built and run as a multiuser task to service many controllers.

A multiuser device handler has the following features:

1 It is task built with the multiuser attribute (/MU MCR switch or /MULTIUSER PDS command qualifer).

2 It has as much code and data as possible in read-only program sections. Such sections are shared when a task is run as multiuser to reduce the amount of resident code.

3 The handlers' TI assignment always refers to the controller that it is servicing, not, as with a single-user handler, to the invoking terminal.

This TI assignment provides a way of associating a particular running copy of the handler with a particular controller. See Section 2.1 for details.

4 The handler declares itself resident in a slightly different way from a single-user handler. See Section 3.1.

1.8 Use Of Ancillary Control Processors

QIO functions can be handled in one of two ways:

1 The handler can perform the required processing by executing code routines within the handler task.

2 They can be directed to an ancillary control processor (ACP) for processing.

The method to be used for each QIO function is indicated by the entry for that function code in the dispatch table within the handler task (see Section 2.2). According to this information, the handler dispatches the QIO either to the appropriate internal routine (using a JMP instruction) or to the ACP task for the device (using a SEND/REQUEST directive). Section Section A.4.3 describes the HNDLIB routine ..DISP, which performs the dispatching.

The ACP method of processing is useful for complicated or multifunction QIOs. Offloading QIO processing to an ACP frees the handler to process other QIOs.

Typically, ACPs perform the following functions:

- File processing operations
- Network protocol operations
- Common I/O database maintenance

2 Device Handler Tables

Each device handler task must set up two tables that are used to communicate with the system subroutines:

- Unit identification table (see Section 2.1)
- Dispatch table (see Section 2.2)

Because the system subroutines use the unit identification table and the dispatch table, both tables must be in the format specified in the following sections.

2.1 Unit Identification Table

Use the unit identification table (UIT) for dequeuing I/O requests. The UIT consists of a 5-word header area, followed by one or more 3-word entries. One 3-word entry is required for each unit serviced by the handler task (for example, a single-unit handler has one entry).

Table 2–1 describes the content of the UIT.

The third column of Table 2–1 shows the values to which the table elements must be preset by the device handler task. The following comments concern word A:

1 For a single-user handler servicing one or more units, word A must be preset to a, b, c ... in successive entries corresponding to the unit number of each unit to be serviced, where a, b, c ... are usually 0, 1, 2 ... . After loading, the handler's TI is assigned to the invoking terminal.

2 For a handler written to drive one unit, and built and run as a multi-user task to service many such units, word A holds a unit validity mask, which must be set to zero.

When the command to load the handler, say for a device type CD, is given, the Executive searches the PUD for all devices CDn. For each CDn, the Executive loads a copy of the handler with its TI assigned to the corresponding unit CDn. The handler code uses this assignment to refer to the appropriate unit.

3 A handler can also be written to service a single controller, which itself governs a number of units. The handler can be built and run as a multi-user task to service many such controllers. The maximum number of units supported by the controller must always be a power of 2. Call this number r.

In this case the first occurrence of word A must be preset to r-1 to form the unit validity mask. Presets of word A for subsequent units are ignored.

When the command is given to load the handler, (for example, EF) the Executive attempts to run a handler for each unit EFn found in the PUD and to run it with a corresponding TI assignment. The system subroutine ..DSMU (see Section 3.1) then compares the mask with the unit number. If the unit number is not a multiple of r, the handler is terminated.

This ensures that handlers can be loaded only as follows:

a copy to service EF0, EF1, ... ,EF(r-1) with TI assigned to EF0

and/or

a copy to service EFr, EF(r+1), ... ,EF(2r-1) with TI assigned to EFr

and so on, as far as necessary.

It also follows that if unit EF(r*j+k), with 0<k<r, has been declared at system generation, it can be serviced only if EF(r*j) is also declared. Thus with r=2 and three units, the units can be EF0, EF1 and EF2. But with EF0, EF1 and EF3 a handler is loaded for EF0 and EF1, but not for EF3.

Table 2–1 Unit Identification Table Contents

Word	Byte	Initial Content	Eventual Content
0	0,1	Address of the Dispatch Table	Unchanged
1	0	Maximum number of units serviced by the handler	Unchanged
1	1	Zeros	Actual number of units to be serviced. Set up by ..DSUT or by ..DSMU
2	0	Zeros	Unit number of the last normal request dequeued. Changed by ..DQRN
2	1	Zeros	Unit number of the last express request dequeued. Changed by ..DQRE
3	0,1	Reserved	Reserved
4	0,1	Reserved	Reserved
		Each Unit Entry Contains the Following Information	
A	0,1	Unit number (single-user handler) or Unit validity mask–multiuser handler; see Section 2.1.	Pointer to start of PUD entry for the unit. Changed by ..DSUT or by ..DSMU.
B	0,1	Zeros	Pointer to start of current normal request node. Changed by ..DQRN when a normal request is dequeued.
C	0,1	Zeros	Pointer to start of current express request node. Changed by ..DQRE when an express request is dequeued.

2.2 Dispatch Table

The dispatch table enables validation of the user task's access rights to the device. Validation is based on the function code of the I/O request and the device class specified.

The dispatch table contains read-only information; its contents are never altered. It contains a 2-word header followed by one 2-word entry for each function code serviced by the handler task. The high-order byte of the function code indicates the order in which the codes occur in the table. For example, the first three entries in a dispatch table could be Special Function (00), Write Logical (01), and Read Logical (02). Table 2–2 provides the layout and content of the dispatch table.

Table 2–2 Dispatch Table Content and Layout

Word	Byte	Content
0	0,1	Address to branch to if a SEND/REQUEST failure occurs.[1]
1	0,1	Address to branch to if SEND/REQUEST is successful.[3].
		Each Function Code Entry Contains the Following Information
A	0	Volume characteristics mask. If set, the bits in word A indicate the following. Bit 7 = The volume must be mounted to perform this I/O function. Bit 6 = The device must be a FILES-11 volume to perform this function. Bit 5 = The volume must not be set for unloading to perform this function. Bit 4 = The volume must be attachable to perform the I/O function. Bit 3 = Device control functions must be permitted to perform this I/O function. Bits 2-0 = Reserved.
A	1	Control variable. If set, the bits in the first word indicate the following: Bit 15 = Word U.DACP of the PUD entry for the device contains the first three characters (in RAD50) of the taskname for the SEND/REQUEST. Bit 14 = The 5 high-order bits of the subfunction code must be 0; otherwise, the request is not valid. Bit 13 = The express bit (bit 1) of the subfunction code must be 0; otherwise, the request is not valid. Bit 12 = Reserved. Bit 11 = The user task must have delete privileges; otherwise, the request is not valid. Bit 10 = The user task must have extend privileges; otherwise, the request is not valid. Bit 9 = The user task must have write privileges; otherwise, the request is not valid. Bit 8 = The user task must have read privileges; otherwise, the request is not valid. Bits 11 through 8 above determine the minimum access rights required, as defined in the volume control block for that user, to allow the user to perform a given I/O function.
B	0,1	If bit 15 of word A for this entry is 0, word B contains the starting address of the routine to process the function code.[3] OR If bit 15 is 1, word B is unused.

[1]Function codes can be processed by the handler (READ/WRITE); or by using SEND/REQUEST to the file processor (OPEN/CLOSE); or by using SEND/REQUEST to Files-11 ACP functions (for example .FIND).

[3]When the second word contains the address of the routine to process the I/O request, it indicates the I/O function code to be executed. If the function is not recognized by the handler (for example, a Create function for a Terminal), the routine addressed by word B immediately returns a valid status (essentially a NOP) or illegal function status by putting the status value in R3 and jumping to the I/O done routine (..IODN). See the discussion of ..IODN in Section A.3.1.

3 Initialization Code

Under normal operation, a newly generated system has only a terminal and a disk handler task initialized and ready to process I/O requests. All other handler tasks must be loaded. The handler task is started at the beginning of the initialization code, which is specified as the transfer address in the same manner as for any other task (for example, .END INIT).

The initialization code must perform the following functions, which are basic to all device handler tasks:

1 Declare the handler task to be resident.

2 Connect to the interrupts.

3 Set up the optional I/O bus activity bit mask (for error logging).

4 Set up the optional I/O statistics buffer (double precision), which must be picked from the system node pool area using either ..PICV or ..PENV (see Appendix A).

5 Perform initialization specific to the device class.

6 Allocate UMRs (UNIBUS mapping registers), if necessary (see Chapter 8).

7 Set up the power failure recovery AST.

3.1 Declaring the Handler Task Resident (..DSUT/..DSMU)

The device handler task calls one of the system subroutines ..DSUT or ..DSMU to declare itself resident and ready to handle I/O requests. ..DSMU is used by multiuser handlers (see Section 1.7), and ..DSUT by non-multiuser handlers. ..DSMU performs some additional checking required by multiuser handlers, then calls ..DSUT. ..DSUT places the active task list (ATL) node address of the handler task in each physical unit directory (PUD) entry corresponding to the device for that handler task. ..DSUT also sets the handler-resident bit in each PUD entry and replaces the unit number in the UIT with a pointer to the first word of the PUD entry for that unit. It also places the virtual address of the UIT entry of the device in offset U.SL of the PUD.

Once ..DSUT has set up the UIT and PUD entries for the device, requests can be queued to the handler task. Prior to this point, the system returned the handler-not-resident status on attempted I/O requests.

The ..DSUT subroutine and the ..DSMU subroutine require the following registers to be preset:

- R0 must contain the address of the UIT.
- R2 must contain the 2-character ASCII device type, for example, TT for the terminal handler.
- R3 must contain the flag byte to be inserted in the PUD flag word. This register is usually set to the global symbol UF.RH implying that the handler task is resident.
- For a disk handler task, UF.TL should also be set (that is, R3 = UF.RH!UF.TL). UF.TL indicates that tasks can be loaded from the device.

After successful execution of the appropriate routine, R1 contains the number of units found, and the C condition code in the Processor Status word is clear. If the routine is not successful, the C condition code is set.

..DSUT or ..DSMU returns an error if:

1 No unit of the specified device type is found.

2 Another handler task is found servicing the unit.

In addition, ..DSMU returns an error if:

1 The device name in the handler's TI assignment does not match the device type declared in R2.

2 The unit validity mask (see Section 2.1) shows that the handler is being run for an illegal unit.

After ..DSUT has returned control to the handler, the handler task should ensure that residency was declared successfully by testing the C condition code in the processor status word. Additional checking might also be useful; for example, the handler could verify the number of PUD entries.

If the handler task is to service a DMA device attached to the UNIBUS of a PDP-11/44 or a PDP-11/70, it must allocate UMRs. Allocation can occur before connecting to an interrupt or dynamically for each transfer. Refer to Chapter 8.

3.2 Connecting to a Vector (..CINT)

Once a device handler task has declared itself resident, it can connect to an interrupt vector. To do so, the handler calls ..CINT. The ..CINT subroutine creates an intermediate program node between the unit's interrupt vector and the device handler's interrupt service routine and calculates the I/O bus activity bit mask based upon the vector address passed to ..CINT as a parameter. ..CINT sets the interrupt vector PC to point to the program node. The program node sets the processor status word to the correct priority, to the desired condition codes for use upon entry to the ISR when an interrupt occurs for the unit, and to Kernel mode.

The ..CINT subroutine must be called once for each interrupt vector serviced by the handler. For example, the terminal handler calls ..CINT twice for each unit serviced: once for the receive interrupt vector, and once for the transmit interrupt vector.

Calling ..CINT locks the handler task into its current memory area. See Section 4.8.3.

If an interrupt service routine is not resident in the same contiguous area of memory as the device handler task (for example, if the interrupt service routine is in an SGA), calling ..CINT fails.

The ..CINT subroutine requires the following registers to be preset:

- R0 must contain the address of the hardware interrupt vector.
- R1 must contain the address of the entry point of the ISR.
- R2 must contain the base address of the ISR area. The base address is usually zero.
- R3 must contain the low-order byte of the Processor Status word (priority and condition codes) available when the ISR begins.

The I/O bus activity bit map enables the system to monitor the traffic on the UNIBUS. The device handler can optionally set the corresponding bit while initiating a data transfer or related function. This bit is automatically cleared in the intermediate program node at the time of interrupt.

If the load on the UNIBUS causes an error to occur, the system error logging capability aids in detecting this type of problem. Error logging uses this to determine all other devices with outstanding requests.

Interrupt service routines can be position-independent code to enable their use under Kernel APR3. However, interrupt service routines are not restricted to position-independent code and its implied addressing mode restrictions. By setting R2 (base of the ISR) to 0, Kernel APR3 maps directly over the user APR0 at interrupt time. Therefore, the relative APR offset for any address while running under Kernel APR3 is 60000(8).

For example, while position-independent code precludes use of an instruction such as the following:

```
MOV ADR(R1),(R2)
```

The ISR can achieve the same result using the following instruction:

```
MOV ADR+60000(R1),(R2)
```

The handler tasks can use the connect to interrupt routine (..CINT) to set condition codes in the PS. Upon entry to an ISR, the condition codes specified in R3 can serve a vitally needed function for multiple-unit handlers with a single ISR. Any number of different interrupts can map into the same ISR. The ISR can use these codes to determine which device has interrupted by examining the four condition code bits (N,Z,V,C) in the Processor Status word. See Chapter 5.

R3 also contains the software priority of the ISR. Normally, it is the same as the hardware interrupt priority. Making the two priorities the same eliminates the need to write reentrant interrupt service routines.

3.3 Device-Specific Initialization

Special initialization code depends on the nature of the device being serviced and varies from handler to handler. For example, the terminal handler task sets the read enable interrupt and selects a read ahead node for each unit it services.

If any errors occur during initialization (usually error exits from system subroutines), the handler task should exit after undoing those functions that have already been successfully performed, as described in Section 4.9.

Normally, it is good programming procedure to make the initialization code the first part of the task such that it is contiguous with the task's stack, and the last instruction of the initialization code a JSR SP,xxx. The designation xxx is the start of the device handling (active) code. Thus, the entire initialization code can be used as stack space. This procedure implies a small initial stack size at task build (link) time and a correspondingly smaller task size at run time.

The handler task build (link) must ensure that sufficient stack space is available. Because handler tasks map the external page, overflowing the stack destroys the content of location 177776 (that is, the processor status word) and usually causes a system crash.

3.4 Setting Up the Power Failure AST

To set up the AST for recovery after power failure, the handler task uses a system directive. The macro SPRA$ defines the address of the routine that performs power failure recovery.

Section 4.7 describes the action you should take on power failure recovery.

4 Handler Task Main Code

The main code of the device handler task dequeues I/O requests and performs the specified input/output operation. The queue contains I/O requests of two types: normal requests and express requests. The difference between normal and express requests is that express requests require immediate attention from the handler (for example, I/O RUNDOWN) whereas normal requests require attention according to their queued priority.

The main code of the handler task performs the following functions:

1 Waits for an I/O completion flag to be set or a request to be queued,

2 Performs the necessary I/O completion functions,

3 Tries to dequeue the next request,

4 Validates the request, if one is dequeued,

5 Dispatches the request to the appropriate I/O function routine, and

6 Exits on request.

Power failure recovery, if present in the handler task, is included in the main code.

4.1 Waiting For I/O Completion or a Request

Among the first steps in the main portion of a device handler task is a wait loop. The loop usually tests for any of the following significant events using the Wait For Logical Or of Event Flags (WTLO$) system directive:

1 Completion of an I/O operation which sets an event flag (normally flag 3),

2 The queuing of an express request which sets event flag 2,

3 The queuing of a normal request which sets event flag 1.

Most of the time it is resident in memory, the handler task is waiting for an event to happen. That wait location is referred to as IDLE in this manual. When an event occurs, the handler must determine exactly what action is required. Therefore, the handler should read its own event flag word and clear the I/O done flag only if an I/O event has completed. The following is an example of the code required:

```
.INH            ;;;INHIBIT INTERRUPTS MACRO
MOV     .CRTSK,R3    ;;;GET TASK'S OWN ATL ADDRESS
MOV     A.EF(R3),R5  ;;;GET EVENT FLAG WORD
BIC     #4,A.EF(R3)  ;;;CLEAR I/O DONE EVENT FLAG
.ENB            ;;;ENABLE INTERRUPTS MACRO
```

Unlike event flags 1 and 2, which are reset by the system dequeuing subroutines (..DQRN and ..DQRE), the I/O done flag must be cleared by the device handler.

After the code illustrated above has executed, the event flags are located in R5. The first flag to be checked by the handler task is the I/O done flag, which is set by the ISR on I/O completion. If the I/O done flag is not set, the handler task tests for an express request and then a normal request, in that order. If I/O done event flag is set, the handler should complete any I/O processing necessary

within the handler and then call the system subroutine ..IODN to notify the requesting task of I/O completion.

All device handler tasks supplied by Digital to date have assumed the following: the ISR is used only to process normal requests; express requests are processed at the handler level. Therefore, the request just completed by the ISR was a normal request. The request node address (RNA) for the completed I/O operation is in word B of the 3-word UIT entry that corresponds to the unit on which the I/O operation occurred. For single-unit handlers, the request node address is in UIT+14. UIT is the starting address of the unit identification table.

For multiple-unit handler tasks, I/O completion processing is more complex than for single-unit handler tasks because an I/O operation may have completed on more than one unit. Therefore, the multiple-unit handler makes three checks for each unit it services:

1 Is word A of the UIT not equal to zero (that is, is the unit present)?

2 Is word B of the UIT not equal to zero (that is, is an I/O operation currently in progress on the unit)?

3 Is the done bit set? The done bit is located in an internal mask word specified for each unit and is set by the ISR upon I/O completion.

If the answer to all three questions is yes, I/O completion must take place for the unit.

4.2 Performing I/O Completion (..IODN)

I/O completion is similar for single-unit and multiple-unit handlers. The handler task calls the ..IODN subroutine to return the request node to the pool and, if specified in the node, to place the indicated I/O status in the user's area and set the event flag or enable the AST indicated by the user for I/O completion.

The ..IODN subroutine requires the following registers to be preset:

- R1 must contain the request node address from UIT word B.
- R2 must contain the adjustment to the decrement for the user's I/O in progress count. R2 is normally 0. Setting R2 to -1 locks the user task in memory and has a function similar to ATTACH. The function similar to DETACH is to set R2 to +1.

  The user task cannot exit while an I/O operation is in progress. Thus, a task attaching a device cannot exit without I/O RUNDOWN, even if the ATTACH request has been processed and returned and no other requests are queued.

To set R2 to anything but zero is an extremely dangerous operation and should only be done if no other solution can be found. The following points should be borne in mind:

1 If a task's I/O in progress count is nonzero, it cannot exit without I/O RUNDOWN being invoked.

2 A task cannot be checkpointed or swapped with a nonzero I/O in progress count.

3 The checkpointing and swapping algorithms assume that:

  - Tasks with I/O in progress can be checkpointed (if they are built as being checkpointable) or swapped (if they are scheduler controlled tasks) when their I/O in progress counts become zero;
  - The I/O in progress counts will become zero in a finite amount of time without the need to dequeue any more I/O requests for the task.

The system therefore prohibits the dequeuing of I/O requests while a task is marked for checkpointing or swapping. Thus, if a task's I/O in progress count is nonzero and it will not become zero except by dequeuing another I/O request, and the task is marked for checkpointing or swapping, then a potentially dangerous situation occurs.

R3,R4 must contain the two I/O status words to be reported to the user. The status words are stored in the requesting task if an I/O status address was provided in the Queue I/O directive parameter block.

Some handlers, particularly those for complex devices (such as overlapped seek disk handlers), might attempt to dequeue requests by calling ..DQRN while a transfer is in progress for a unit. Such handlers should unconditionally call ..DQRN (or equivalently should set their own normal request event flag, event flag 1) after performing I/O done processing. This is true because requests might be queued for that unit while it is busy. In this case ..DQRN will have indicated, correctly, that no requests could be dequeued. Now that requests can be dequeued, it is necessary to indicate that the situation has changed by calling ..DQRN.

This complication does not apply to handlers that do not attempt to dequeue normal requests while performing a transfer.

4.3 Dequeuing a Request (..DQRN and ..DQRE)

When an I/O request has been queued, the handler task determines whether the request is normal or express by looking at its event flags. If an express request is indicated (flag 2), the handler calls the system subroutine ..DQRE. If a normal request is indicated (flag 1), the handler calls ..DQRN. If neither flag is set, the handler should return to its IDLE code to await further events.

Because ..DQRN performs the same functions for a normal request as ..DQRE does for an express request, they are described together in this section. Both should be coded using the same path. The following information pertains to both subroutines:

1 Both subroutines attempt to dequeue the highest priority request in the queue (normal or express). However, ..DQRN is restricted if a user task is attached to the unit; only requests from that task or from tasks with a privileged UIC (group code less than 10) are dequeued.

2 Both subroutines dequeue requests only for a unit whose UIT request node address (RNA) is zero. For ..DQRN, the RNA is in word B of the unit entry. For ..DQRE, the RNA is in word C of the unit entry. See Table 2-1.

3 Both subroutines attempt to take the request node from the queue, place the request node address in the UIT entry word (either B or C) and in R1, and place the address of word A of the unit's UIT entry in R2.

4 If the attempt in 3 above is unsuccessful, ..DQRN and ..DQRE attempt to handle other units, if any, associated with this handler. If they cannot dequeue any requests, they set the C condition code in the Processor Status word and reset the appropriate event flag (1 for ..DQRN and 2 for ..DQRE).

  The subroutines ..DQRN and ..DQRE require R0 to be preset to the starting address of the UIT. Once set to the starting address, R0 should remain unaltered.

The dequeuing subroutines cannot be called from the interrupt service routine.

The exit conditions are as follows:

- R0 - Is the address of the UIT.
- R1 - Is the address of the request node or is undefined if no node is found.

- R2 - Is a pointer to the PUD pointer in the UIT or is zero if no node is found.

4.4 Validating a Request (..VACC)

Once an I/O request is dequeued, the handler task must validate it. The validation process consists of two steps: handler task validation of the function code, and validation of the ..VACC subroutine.

Because ..VACC validates the I/O request based only on the contents of the dispatch table, the handler task must validate the function code to see if it falls within a legitimate range (normally 0 through 27). If the function code is not in the legal range, the request node must be returned to the system by means of the ..IODN subroutine and an error status must be set for the user in R3.

If the function code is within range, the handler task calls ..VACC. The following registers (that have already been initialized by either ..DQRN or ..DQRE) are passed to ..VACC:

- R0 contains the address of the UIT.
- R1 contains the request node address.
- R2 contains a pointer to word A of the unit's UIT entry.

To validate the user's right to issue the I/O request, ..VACC examines the following:

1 The volume control block for the unit,

2 The handler's dispatch table entry for the function code specified, and

3 The user's UIC.

If validation of any of the above checks fails, ..VACC returns to the handler with the C condition code set, and the handler returns an error code to the user. If the request is valid, ..VACC returns the C bit as zero and the handler task then jumps to the dispatch routine (..DISP).

4.5 Dispatching the Request (..DISP)

Unlike all other system routines, ..DISP is not a subroutine; the handler task must execute a JMP instruction, rather than a JSR, to reach it. ..DISP uses the dispatch table to determine the I/O processing routine or the SEND/REQUEST function to be performed.

The following registers must be preset:

- R0 must contain the address of the UIT.
- R1 must contain the request node address.
- R2 must contain a pointer to word A of the unit's UIT entry.

R0 acts as a pointer to the dispatch table's address, which is stored in the first location of the UIT.

At this point, the I/O request has been dequeued, validated, and routed to the I/O processing routine that performs the specific function requested.

4.6 Processing the I/O Functions

The host operating system provides numerous I/O function codes, most of which are serviced by one of seven I/O processors. Normally, an I/O processor can handle many codes for various device handler tasks. The processor to be selected is indicated in the function code. The *IAS Device Handlers Reference Manual* lists the function codes.

The seven I/O processors presented in this manual are as follows:

I/O Processor	Code
Special Functions	0
Write Logical	1
Read Logical	2
Attach	3
Detach	4
NOP	Any legal code that requires no special action by the handler task.
Error	Any illegal code for the device.

Of the seven processors, only Write Logical and Read Logical require interrupt service routines for execution; for these two, I/O is initialized. The other I/O processors and any error conditions that are detected require a separate path for I/O completion (that is, a call to ..IODN). Since no event flags are set for the special functions, Attach, Detach, NOP, and Error processors, the I/O completion path is identical to that described in Section 4.2, with the following exceptions:

1 The normal request's queued event flag does not need to be set again because ..DQRN or ..DQRE was not called after the current request was dequeued.

2 The request node address (RNA) in either word B or word C of the unit's UIT entry, depending on whether the request is normal or express, must be cleared if dequeueing is to proceed. Therefore the function code of the current request must be saved, the ..IODN subroutine called, and the following code executed:

```
MOV (SP)+,R3     ;ASSUME FUNCTION CODE WAS ON STACK
BIC #177775,R3    ;CLEAR ALL BUT EXPRESS BIT
TST (R2)+       ;ADVANCE R2 FROM FIRST WORD OF UIT
          ;TO 2ND WORD OF UNIT ENTRY
ADD R3,R2       ;IF FUNCTION JUST PROCESSED WAS
          ;EXPRESS R2 IS NOW AT THE 3RD
          ;WORD OF UIT ENTRY
CLR (R3)        ;CLEAR RNA AND REENABLE DEQUEUER
```

4.6.1 Special Functions Processor

The special functions processor, indicated by function code 0, handles three different subfunctions:

1 KILL ALL REQUESTS (subfunction code 12),

2 I/O RUNDOWN (subfunction code 22), and

3 UNLOAD HANDLER (subfunction code 42).

All three are express requests (bit 1 of the subfunction code is set) and are dequeued accordingly.

A user task issues KILL ALL REQUESTS to a logical unit to cancel all requests from that task that are incomplete on that unit and to clear the attach bit in the device's PUD entry if it is set.

I/O RUNDOWN occurs whenever a task exits or is aborted with I/O requests pending. Since a task's memory cannot be reused while data transfers still can be initiated by I/O requests that have not been dequeued, the system issues an I/O RUNDOWN node sequentially to each unit defined in the PUD. The node identifies the task for which the rundown is to occur. The handler task aborts all I/O request nodes that remain in the task's queue and usually terminates any requests in progress for the task. This process is essentially the same as KILL ALL REQUESTS.

The system issues a request to unload the handler whenever the MCR UNLOAD or the DCL STOP/HANDLER commands are used. The handler task is expected to complete all current requests and then exit. New requests are prevented from being queued.

4.6.1.1 KILL ALL REQUESTS and I/O RUNDOWN

Because KILL ALL REQUESTS and I/O RUNDOWN are similar in function, their respective function processors should handle them identically, with the following exceptions:

1 The I/O RUNDOWN request node should be checked to ensure that it came from the Executive and not from a user task. If RNA is in R1, R.AT(R1) contains the ATL address of the issuing task. It is zero if the Executive issued the request.

2 The KILL ALL REQUESTS node from a user task must have its parameter words set identically to those of an I/O RUNDOWN request node. The three parameters that must be set by the handler follow:

   Parameter word 1 = active task list (ATL) address for the task,

   Parameter word 2 = system task directory (STD) address for the task, and

   Parameter word 3 = starting address of PUD entry for the unit.

Once the parameter words are set, the handler task can call the system subroutine ..FLSH to remove all I/O requests for the specified task from the queue. Now the handler task can return the express request node to the node pool. It also can check for the following events:

1 Is there an I/O operation currently in progress on the unit?

2 Was the operation requested by the user task?

If the response to both checks is yes, and if the I/O operation can take a significant time (for example, a read operation on a terminal), the handler task should use the following procedures:

1 Terminate the I/O operation. For example, for a terminal you should set the characters received to equal the maximum-characters-asked value (to prevent further reads) and set the write enable bit.

2 Return the request node of the I/O operation. In order to return the node, the following information in the node must be zeroed before calling ..IODN:

   The I/O status block buffer address,
   The event flag word, and
   The AST word.

   Resetting this information to zero prevents ..IODN from altering any status bits associated with the task. This makes it appear that the node for the I/O operation in progress was also flushed.

The RNA of the I/O operation in progress is located in word B of the UIT entry for the unit.

4.6.1.2 UNLOAD HANDLER

When a device handler receives an UNLOAD HANDLER request, it should check first to ensure that the request comes from the Executive (that is, if RNA is in R1, R.AT(R1) = 0). If the call is not from the Executive, an error code must be returned. If it is from the Executive, the handler task should set an exit flag for itself and return the node.

The IDLE code (that is, that section of code executed when the handler is waiting for requests to be queued) should contain a check to determine whether the exit flag is set. This check should precede the system directive to wait for multiple event flags. When the flag is set, the IDLE code should check every unit to see if any I/O operations are in progress (word B of every UIT entry should equal zero). If no I/O operations are in progress, the handler task should branch to its exit code; otherwise, it should execute the WAIT FOR directive. In this manner, the handler task exits only when no I/O operations are in progress on any unit.

4.6.2 Read Logical and Write Logical Processors

The servicing of Read Logical and Write Logical I/O functions essentially is device dependent; however, strong similarities exist in the handling of the two functions in different handlers. A typical processing scheme is described in the following paragraphs.

Both read and write functions usually have a user buffer where data is stored or retrieved. The buffer must be validated to determine whether the entire buffer is in the user's area. By convention, the first two parameter words in a read or write I/O request node define the buffer in the following format:

- Parameter word 1 = start of the buffer in the user's virtual area, and
- Parameter word 2 = size of the buffer specified in bytes.

A handler for a non-DMA device must, after checking the buffer, set up three internal locations for use by the ISR:

- UBASR = APR value for the start of the buffer,
- UBSAP = offset of the first location in the buffer from the APR address, and
- UBCAP = offset of the current location in the buffer from the APR address.

Using these locations, the ISR replaces the current address and descriptor registers for Kernel APR2 with UBASR and 77406 (4K read/write segment), respectively. Thus, setting the three locations directly maps the ISR into the user's buffer.

An ISR that uses (Kernel) APR2 must first save the contents of the corresponding memory management registers and afterwards restore them.

The handler task calls the ..VXFR system subroutine to set the locations. It requires the following preset registers:

- R1 must contain the RNA (already set).
- R2 must contain the starting address of the buffer (parameter 1 of the RNA or R.PB(R1)).
- R3 must contain the number of bytes in the buffer (parameter 2 of the RNA or R.PB+2(R1)).
- R5 must contain the type of validation (0 = read function, 1 = write function).

If the user buffer is correct, the output of ..VXFR is the following:

1 R4 set to the high-order two bits of the absolute address of the start of the user's buffer in bits 4 and 5 of R4,

2 R5 set to the low-order 16 bits of the absolute address of the start of the user's buffer, and

3 The C condition code is set only if the buffer was not valid.

For DMA transfers, the output of ..VXFR provides the correct 18-bit address; no further action is required to obtain the address.

For non-DMA transfers, however, the following code must be executed to isolate the APR address and offset values in the locations specified above:

```
ASH     #-4,R4      ;SHIFT 2 HIGH ORDER BITS TO BITS 0,1
ASHC    #12,R4      ;ISOLATE OFFSET IN R5, APR ADDRESS IN R4
ASH     #-12,R5     ;RIGHT JUSTIFY OFFSET
BIC     #177700,R5    ;MASK OFF EXCESS
ADD     #40000,R5   ;SET IT AS APR2 OFFSET VALUE
MOV     R4,UBASR    ;SET KERNEL APR2 ADDRESS REGISTER
MOV     R5,UBSAP    ;SET KERNEL APR2 STARTING OFFSET
MOV     R5,UBCAP    ;SET KERNEL APR2 CURRENT OFFSET
```

Once the code above executes, the ISR has all the necessary values. After performing any specialized device functions (for example, turning on an interrupt enable bit), the handler task should attempt to service further events.

Note that special code must be executed for PDP-11/44 and PDP-11/70 UNIBUS devices; refer to Chapter 8.

4.6.3 Attach and Detach Processors

When a user task issues an attach or detach request, the attach or detach processor calls the appropriate system subroutine, that is, ..ATUN or ..DTUN respectively. All the necessary registers are already preset by ..DQRN as follows:

- R1 contains the request node address.
- R2 contains a pointer to the PUD address in the UIT.

If an error occurs, an error code should be returned. If no error occurs, the handler task should issue a success code. In either case, immediately after the device is attached or detached the node must be returned by using the ..IODN subroutine. The handler task then attempts to dequeue further requests.

4.6.4 NOP and Error Processor

The NOP and error processors perform essentially the same functions. Both return the node using ..IODN and set user status information. The NOP processor indicates that the function was successful, and the error processor indicates the cause of the error.

4.7 Recovering From Power Failure

If the handler task contains the system directive to specify a power failure AST entry point (SPRA$) in the initialization section, the power recovery section of the handler is executed when either of the following conditions occurs:

1 The system is bootstrapped with the handler already loaded, and

2 Whenever a power recovery occurs.

The function of the power failure code is to enable system operation to continue as though nothing had happened, even if there was an I/O operation in progress on the device at the time. Typically this will involve:

1 Perform a controller clear, in case the power failure did not affect the device.

2 Wait for mounted devices and those with I/O in progress to become ready again (for example, to reach full speed). The system subroutine ..PWUP (see below) is available to simplify this process.

3 Restart any transfers which were in progress. For simple devices it may be necessary only to set the interrupt enable bit. For more complex devices the action required depends upon the device.

The subroutine ..PWUP in the handler library can be used to wait for mountable devices to become ready. When called, it does not return until all mounted units have become ready, or until a specified timeout period has elapsed in case a unit has been turned offline. The following registers must be set up first:

- R0 must contain the timeout period in seconds, and should be greater than the maximum time for a drive to become ready under normal conditions.
- R1 must contain the address of a user-written subroutine which checks to see if a unit is online (see below).
- R2 must contain the address of the UIT.

The routine whose address is given in R1 is called once a second for each mounted unit to see if it is ready. It is given the PUD address of the unit in R0, and should return with C condition code set if the unit is not ready or clear if it is.

Because a power fail AST occurs when the system is booted, possibly on a different configuration, it must be able to deal with certain changes in the configuration. In particular, it must check to see whether UMRs are needed (for handlers to DMA devices) and allocate UMRs if necessary. For example, a system can be saved on a machine with less than 124K words and booted on a larger machine, and the reverse can also happen.

4.8 Swapping Considerations

Since most device handler tasks transfer data to or from an I/O buffer that resides within the requesting task, that task cannot be swapped until the I/O operation is complete. If the transfer is performed to or from a buffer outside the requesting task (for example, a buffer residing in the handler task), the requesting task can be swapped while the requested I/O operation is in progress. Also, the task can be freed if the request does not include data transfer, for example, tape rewind. The system routines described below are used to free a task for swapping and to lock a task in memory.

4.8.1 Free a Task for Swapping (..FRSW)

This routine releases a task for swapping while I/O is in progress. ..FRSW is normally called after the handler task has obtained all required information from the requesting task's memory space. Refer to Appendix A.

R1 must contain the request node address before calling ..FRSW.

4.8.2 Get Task Back in Memory (..TKBK)

This routine brings a task back into memory that was previously released for swapping via ..FRSW. The routine ..TKBK is normally called when the handler task wishes to complete a requested I/O operation and transfer data to the requesting task. The task does not need to be in memory for the handler to perform I/O done.

R1 must contain the address node before calling ..TKBK.

If the requested task is not in memory, ..TKBK marks the task for reloading and returns with condition code C set. Event flag 3 is set when the task is back in memory. The handler should repeat the call to ..TKBK to ensure that the task is locked in memory as indicated by a return with condition code C clear. Thereafter, the handler can resume I/O processing. Refer to Appendix A.

For slow transfers, the handler must buffer the data or release the task to prevent the locking of memory and loss of response time for other tasks in the system.

It is important that ..TKBK be called only for a task which has been freed by calling ..FRSW. Calling ..TKBK for a task which is already locked in memory will cause subsequent swapping to behave unpredictably.

4.8.3 Locking a Handler Task in Memory

In IAS, calling the routine ..CINT (Section 3.2) within the handler task locks it into the current memory area where it is loaded. In particular, if the handler is in a timesharing type partition it is not shuffled (moved towards the low end of the partition to create contiguous free memory).

4.9 Exiting From the System

The handler task should execute its exit code when either of the following conditions occurs:

1 A request to unload the handler task has been dequeued and all I/O operations requested, if any, have finished, or

2 A system subroutine fails during handler initialization.

Normally the exit code performs the reverse of those functions contained in the initialization code:

1 It performs device-specific processing,

2 It disconnects from the interrupts,

3 Returns (deallocates) UMRs, if necessary,

4 It declares the handler task nonresident, and

5 It exits from the system.

4.9.1 Device-Specific Exit Processing

Device-specific exit processing consists of those functions that effectively turn off the device for the handler task. These functions include turning off the interrupt enable bits and releasing the nodes used for internal processing of I/O requests for all units serviced by the handler.

4.9.2 Disconnecting from Interrupts

Each interrupt to which the handler task connected using ..CINT must be disconnected. Disconnection is accomplished by calling the ..DINT subroutine with R0 set to the interrupt trap vector address to be disconnected. That address is located in offset U.TV within the PUD entry for the unit. The subroutine returns the code node created by ..CINT to the pool and resets the trap vector PC to the undefined-interrupt-seen Executive routine. If the reason for exiting is an error return from ..CINT, the handler task must not attempt to disconnect itself from that interrupt, but should disconnect from any other interrupts successfully connected.

4.9.3 Declaring Nonresidency

The handler task calls the ..DNRC subroutine to declare itself nonresident. R0 must contain the address of the handler's UIT. The subroutine prevents further interaction between the handler task and the system by performing the following functions for each unit of the UIT:

1 Removing all undequeued I/O requests and returning them to the pool using ..IODN,

2 Clearing any attached task ATL pointer from the unit's entry in the PUD, and

3 Clearing the handler's ATL pointer from the unit's PUD entry and clearing the handler-resident bit.

If the reason for exiting is an error return from ..DSUT during initialization, the handler task must not declare itself nonresident.

4.9.4 Handler Task Exiting

The handler task exits from the system by using the Exit system directive. If the exit code is correct, no I/O operations are pending, no nodes remain in use, and the handler's memory area can be freed for subsequent use. The handler task can be reloaded into memory at any time.

Note that if a handler task exits without declaring itself non-resident, a system crash is almost inevitable. This is because the executive maintains in the PUD the real memory address of the handler task. If the executive performs an I/O rundown for any reason (for example, a task is aborted) it accesses the addresses of all handlers which it believes to be resident. The executive cannot recover if the address is occupied by other code.

5 Interrupt Service Routine

The interrupt service routine (ISR) is executed whenever an interrrupt occurs for a vector to which the handler is connected. All interrupt service routines execute in kernel mode using APR3. Therefore, you must observe the following rules:

1 Save ane restore all registers used in the ISR.

2 The ISR cannot exceed 4K words without incurring mapping difficulties.

3 The ISR code must be position-independent or the APR3 offset (60000) must be taken carefully into account.

4 The only available APR to use when mapping into the user's buffer is APR2 because the kernel set of APRs is being used.

Note that the Executive maps into the Kernel APR2. Therefore, a handler that uses Kernel APR2 must always save and restore that register.

The following rule applies only to multiple-unit handler tasks: on entry to the ISR, the processor status Word (PS) must be saved because it contains the unit number of the interrupting device in the condition code bits. Therefore, the first few instructions of a multiple-unit handler perform the following:

```
MOV  @#177776,-(SP)    ;SAVES THE PS (AND UNIT #)
MOV  R4,-(SP)        ;SAVE R4
MOV  2(SP),R4        ;GET UNIT # IN R4
BIC  #177760,R4       ;ISOLATE UNIT #
MOV  R5,2(SP)        ;SAVE R5
MOV  R3,-(SP)        ;SAVE R3
```

The ISR also must set the I/O done event flag (flag number 3) of the handler task when the current interrupt completes an I/O request. To set the flag, the handler task calls the ..STEF subroutine which requires the following preset registers:

- R0 must contain the address of the PUD entry for this unit.
- R1 must contain the bit pattern of event flags to be set (bit 2, in this case).

Assuming that the three locations named UBSAP, UPCAP, and UBSAR have been initialized by the function processor code as specified above, the ISR can obtain the next byte in the user's data buffer through the following code sequence:

```
MOV  @#172304,-(SP)    ;SAVE CURRENT APR2 DESCRIPTOR ARG
MOV  @#172344,-(SP)    ;SAVE CURRENT APR2 ADDRESS ARG
MOV  #077406,@#172304   ;SET APR2 FOR 4K R/W
MOV  UBASR,@#172344    ;SET APR2 ADDRESS TO USER BUFFER
MOV  UBCAP,R5        ;SET R5 TO CURRENT BYTE IN DATA
```

UBCAP contains the 40000 offset necessay to map it through APR2 when an indirect reference is given. Use mnemonics instead of the octal numbers.

Once the interrupt is serviced, the following exit sequence is required:

1 Increment UBCAP to the next byte in the user's buffer.

2 Restore Kernel APR2 address and descriptor registers.

3 Restore all used registers to their values before the interrupt, and

4 Jump to the Executive EXIT ISR routine ..INTX using the following instruction:

```
JMP @#..INTX
```

The ..INTX subroutine restores the kernel APR3 values and returns control to the interrupted task.

Do not include the interrupt service routine in a read-only program section, if the handler has one. This is because the read-only part of a task uses a separate APR and will not be mapped into APR3 when the interrupt occurs.

6 System Generation and Task Building

For a device handler task to be debugged and used, a system must be generated to include the device that the handler task services. The handler task is incorporated into the system by linking the task using the task builder, then installing the handler task in the system.

6.1 System Generation Requirements

The device directive (DEV=) defines peripheral devices to the host operating system. The format of the device directive for standard devices is detailed in the *IAS Installation and System Generation Guide*. The format of the device directive for peripherals not supported by Digital is as follows:

- DEV = device mnemonic,unit type,trap,priority, ext page addr,acp
- device mnemonic = 2 ASCII characters indicating the device class followed by the unit number, for example, TT1.
- unit type = 4 characteristics words to describe the unit. Each word is specified as an octal value. Words are separated by commas. (See Table 6-1.)
- trap = interrupt vector address.
- priority = processor priority
- ext page addr = address of the first external page for the unit.
- acp = ancillary control processor.

Using the information specified in the device directive, system generation creates an entry in the physical unit directory to contain device information. The format of the PUD is specified in Appendix B.

Table 6–1 Device Directive Unit Type Characteristics Words

Word	Bit	Offset	Meaning If Set (Bit=1)
1	0	UC.REC	Indicates a record-oriented device, for example, card reader.
	1	UC.CCL	Indicates a carriage control device, for example, line printer.
	2	UC.TTY	Indicates a teleprinter-like device, for example, LA30.
	3	UC.DIR	Indicates a multiple directory device.
	4	UC.SDI	Indicates a single directory device.
	5	UC.SQD	Indicates a sequential device, for example, magtape.
	6	UC.IAS	Indicates an interactive terminal (timesharing systems only).
	7	UC.IEX	Indicates an exclusive device (for timesharing systems only).
	8	UC.INB	UNUSED
	9	UC.SWL	Indicates that the device is software write locked.
	10	UC.ISP	Indicates an input spooled device.

Table 6–1 (Cont.) Device Directive Unit Type Characteristics Words

Word	Bit	Offset	Meaning If Set (Bit=1)
	11	UC.OSP	Indicates an output spooled device.
	12	UC.PSE	Indicates a pseudo-device.
	13	UC.COM	Device is communications channel.
	14	UC.F11	Device is Files-11.
	15	UC.MNT	Device is mountable.
2&3			Contain the device-specific special unit characteristics flags used by the handler. See Section 4.6 of the *IAS Device Handlers Reference Manual* for the values to use for disk-type devices.
4			Contains the maximum block size for the device, for example, 132 bytes for a wide line printer and 512 bytes for a disk.

6.2 Linking

Use the task builder is used to create an executable task image. Remember the following points when you build a device handler task:

1 The task must be executive privileged. That is, it must be built with the switches /PRIVILEGED (DCL LINK command) or /PR (MCR TKB command).

2 Access is required to the Handler Library subroutines contained in HNDLIB. HNDLIB is a Shareable Global Area (SGA), and, since it is required by the system disk handler, is normally resident in memory. The HNDLIB SGA is accessed by including the Task Builder option SGA, for example,

```
SGA=HNDLIB:RO:2
```

HNDLIB is position-independent and can be mapped into either APR1 or APR2. The above line places it in APR2 which should normally be used. APR4-7 are not available as they are used to access SCOM and the external page. APR3 must not be used because many of the routines in HNDLIB employ the APR3 page for work space.

Some very large handler tasks cannot spare an APR to map on to HNDLIB. In this case such routines as are needed can be extracted from the object module library [11,14]HNDLIB.OLB. They are included in the task image via the input file specification

```
[11,14]HNDLIB/LIBRARY
```

in a DCL LINK command or

```
[11,14]HNDLIB/LB
```

in an MCR TKB command.

This saves virtual address space but increases the task's real memory requirement by .75K to 2K, depending on which routines are called. If this method is used, the build command must include [1,1]EXEC.STB. (If the SGA method is used, EXEC.STB need not be referenced explicitly in the build command as all symbols are defined in HNDLIB.)

3 A device handler must be non-abortable and non-checkpointable.

4 The handler task name must be 'xy....', where xy is the type of the device to be serviced, for example 'DK' for the RK05 handler.

5 A device handler should normally run under the system UIC of [1,1].

6 A handler will normally require only a very small stack, since the initialization code is normally overwritten to provide stack space. 32 (decimal) words of stack will usually be sufficient.

7 A multi-user device handler should have all its pure data and all its code except its ISR and its initialization code (see step 6 above) in read-only p-sections.

8 Device handlers will not normally use the Floating Point Processor. They should be built /-FP (MCR TKB command) or /NOFLOAT (DCL LINK command) to avoid saving and restoring the floating point registers at each context switch.

6.2.1 Examples of Build Files

DCL LINK command specifying the handler routine resident library:

```
$LINK/NOABORT/NOCHECKPOINT/PRIVILEGED/NOFLOATING-
/OPTIONS XY
TASK=XY....
UIC=[1,1]
STACK=32
SGA=HNDLIB:RO:2
/
```

DCL LINK command specifying the handler routine object module library:

```
$LINK/NOABORT/NOCHECKPOINT/PRIVILEGED/NOFLOATING-
/OPTIONS XY,[11,14]HNDLIB/LIBRARY,[1,1]EXEC.STB/SELECT
TASK=XY....
UIC=[1,1]
STACK=32
/
```

MCR TKB command specifying the handler routine resident library:

```
XY/-AB/-CP/PR/-FP=XY
/
TASK=XY....
UIC=[1,1]
STACK=32
SGA=HNDLIB:RO:2
/
```

MCR TKB command specifying the handler routine object module library:

```
XY/-AB/-CP/PR/-FP=XY,[11,14]HNDLIB/LB,[1,1]EXEC.STB/SS
/
TASK=XY....
UIC=[1,1]
STACK=32
/
```

See the *IAS Task Builder Reference Manual* and the *IAS PDS User's Guide*.

7 Error Logging

7.1 Introduction

Error logging is performed by three routines that gather information on device errors that occur and produce a report from that information. This is accomplished by the tasks ERRLOG, PSE, and SYE.

ERRLOG is a preinstalled task that gathers volatile information when a device error occurs, and places it in a temporary file called ERR.TMP, in UFD [1,6] on the system, or user-selected device. When a report of device errors is desired, the other two tasks are run.

The preanalyzer, PSE, uses the information in the file created by ERRLOG to produce a formatted file for use by the analyzer task, SYE. PSE will call ERRLOG which will rename ERR.TMP to ERROR.TMP, delete ERR.TMP and open a new ERR.TMP to continue logging errors. PSE will use the information in ERROR.TMP to create a new, formatted file called ERROR.SYS. Once the formatted file has been created, ERROR.TMP is deleted.

The analyzer task, SYE, produces from ERROR.SYS a list file whose name and content depend on the options specified by the user. ERROR.SYS is not deleted.

7.2 Error Log Support for Device Handlers

Error log support for device errors requires that the system do the following:

1 Maintain an I/O active-bit map

2 Detect hardware errors as they occur

3 Maintain statistics on device errors.

The I/O active-bit map depicts the state of the system at the time an error was detected. The active-bit map is keyed to the interrupt vectors, with a bit being allocated to each of the 128 possible vectors. When an I/O request is issued (the "GO" bit is set) the bit corresponding to the vector that this device traps to on completion is set to 1. When the interrupt is detected, this bit is automatically cleared.

The detection of device handler errors occurs at the interrupt service routine (ISR) level.

Device statistics are kept on a per-unit, per-device basis.

7.3 Error Logging Interface

This section covers the following topics:

1 Handler initialization (see Section 7.3.1).

2 Loading the function register (see Section 7.3.2).

3 Interrupt Service routine (see Section 7.3.3).

4 The MOUNT command (see Section 7.3.4).

5 Handler exit (see Section 7.3.5).

7.3.1 Handler Initialization

If the call to ..CINT is successful, the active-bit mask (global offset I.MK) and the word location (global offset I.MD) must be saved if they are to be used later. The code node address exists in word 0 (Virtual), and the word location exists in word 2 (Virtual) of the handler task. Save the bit mask and word location by using the following sequence of instructions:

```
MOV @#0,R1          ;GET CODE NODE
                    ;ADDRESS FROM
                    ;VIRTUAL 0

MOV I.MK(R1),BMSK     ;SAVE BUS
                    ;INTERACTION
                    ;MASK

MOV I.MD(R1),BWD     ;SAVE ADDRESS
                    ;MASK WILL
                    ;BE APPLIED
                    ;TO
```

Also, a statistic node, with a block of words two times the number of units, must be picked from the system pool area.

7.3.2 Loading the Function Register

Just before the device function register is loaded, the saved bit mask and word location must be used to set the appropriate bit in the I/O active-bit map. This can be accomplished by the following instruction:

```
BIS  BMSK,@BWD      ;SET INTERACTION BIT
```

Also, the appropriate double precision statistic count must be incremented for the appropriate device unit number.

The global offset for the PUD pointer in the RNA is R.PD.

7.3.3 Interrupt Service Routine

When the ISR detects a device error, a call to the system common subroutine ..ERLI must be made with the following registers preset:

- R1 = the request node address
- R3 = the number of device registers to be passed
- R4 = the beginning address of the double precision device block
- R5 = maximum retry count (high-order byte); current retry count (low-order byte)

If the call to ..ERLI is successful (that is, the "C" bit is not set), then registers 0 through 4 are unchanged and register 5 contains the starting address within the dynamic buffer which the handler will use to pass a maximum of 30 of its device registers.

The unit descriptor word value (see Table 7–2) is inserted. The global offset for this value is .ELOF(R5).

If the call to ..ERLI is not successful, the "C" condition bit is set on return to indicate an error, and the contents of R5 are undefined. An unsuccessful call to ..ERLI may be the result of:

1 An error was already detected for the same QIO request (that is, this is a retry).

2 ERROR LOG is not active.

3 No dynamic buffers are available at the time of error.

7.3.4 MOUNT Command

Whenever a device is mounted, the appropriate double precision count must be cleared for the appropriate unit (See Table 7–1).

7.3.5 Handler Exit

Immediately after the call to ..DINT, the statistics count block node must be released to the system node pool.

7.4 Errlog Task Responsibility

This section covers the following topics:

1 ERRLOG task initialization (see Section 7.4.1).

2 ERRLOG task processing (see Section 7.4.2).

7.4.1 ERRLOG Task Initialization

The user must select an 80-word (decimal) node for each value entered in response to the following message:

```
INPUT NUMBER OF ERROR BUFFERS "CARRIAGE RETURN"

THIS VALUE SHOULD BE BETWEEN 1 AND 5. IF ERROR LOGGING NOT
WANTED, INPUT CONTROL Z.

NUMBER OF ERROR BUFFERS =
```

7.4.2 ERRLOG Task Processing

Set up a five-second mark time AST to guarantee the logging of any device error within a five-second period.

Wait on event flag 8 from a handler task or event flag 64 (decimal) from the PSE task. When event flag 8 is set, create the file, ERR.TMP (see Table 7–1 for the format).

When event flag 64 is set from the PSE task, rename ERR.TMP under [1,6] to ERROR.TMP on the logging device.

If at any time a device handler error occurs that is caused by the ERRLOG task, the ERR.TMP file is renamed ERROR.TMP and the appropriate error message, with the standard system error codes, are output to CO. All 80-word (decimal) ERRLOG nodes are released to the system node pool. The device handlers are notified by the setting of the "C" bit when they go to log an error that the ERRLOG task has exited.

Table 7–1 Record Format* of ERR.TMP and ERROR.TMP Files for Device Errors

Word	Definition
0	Number of words in this record
1	Record number
2	Cumulative error sequence number
3	Number of device registers
4	Device name, 2 ASCII characters
5	Unit number (low-order byte)
6	Unit descriptor word (low-order byte) (see Table 7-2.)
7	RK05 Error count
8	Type of error (Only 1 = device error is valid)
9	Year
10	Month
11	Day System date and time of error
12	Hour
13	Minute
14	Second
15	First word of erring task name
16	Second word of erring task name
17	Requestor UIC
18	1/64th of real address of load image
19	I/O function code
20-25	I/O parameters
26	Handler maximum retry count (high-order byte); handler current retry count (low-order byte)
27	Requesting task I/O in progress count (low-order byte)
28	Requesting task I/O pending count (low-order byte)
29-34	Volume label (ASCII)
35	Volume UIC
36	First word of erring unit double precision count
37	Second word of erring unit double precision count
38-41	Not used
42-49	I/O active-bit map
50-79	Erroring device register contents (up to 30 (decimal) device registers)

*The 80-word description of the record format is preceded by two linkage pointer words for a total of 82 words for each node.

The unit descriptor word is divided into two parts. Bits 0 to 3 contain the device class value, bits 4 to 7 contain the device type value.

Table 7–2 Unit Descriptor Words

Class (bits 0 to 3)	Value (bits 4 to 7)	Meaning
	Device classes and values	
Disks (1)	1	RK05
	2	RP03
	3	RF11
	4	RS04
	5	RS03
	6	RP04
	7	RP02
	10	RK06
	11	RP05
	12	RP06
	13	RK05F
	14	RK03
	15	RX01
	16	RM03
	17	RL01, RL02
	20	RX02
	21	reserved
	22	RM05
	23	reserved
Tapes (2)	1	TU56
	2	TU10
	3	TU16
	4	TU11
	5	unused
	6	reserved
	7	TU58

8 Special Considerations For DMA Devices

You must consider the information contained in this chapter when you write a device handler task for a device that performs DMA transfers.

In general, this is important only if the device is one of the following:

1 A UNIBUS device attached to the UNIBUS of an 11/70 which is using more than 124K words of memory.

2 A UNIBUS device attached to the UNIBUS of an 11/44 which is using more than 124K words of memory.

3 A MASSBUS device attached (using an RH11 controller) to the UNIBUS of an 11/44 using more than 124K words of memory. (NOTE - There is no MASSBUS on an 11/44.)

However, if you use certain routines, UNIBUS and MASSBUS handlers can be written so that they can be run on the configurations listed above and alternate configurations without change to the handlers' code. Sections 8.3.1.2 and 8.3.2.2 describe these routines more fully.

8.1 Introduction to UMRs

The PDP-11/44 and PDP-11/70 have memory systems that are not connected directly to the UNIBUS. The memory system also can be larger than that of other PDP-11 systems. When using more than 124K words of memory, the PDP11/44 and PDP-11/70 CPUs operate in 22-bit addressing mode, using all 16 bits of the PARs in the memory management unit. This mode is enabled in memory management register 3. Refer to the *PDP-11 Processor Handbook* for complete information on extended addressing.

The CPU has its own path to the memory system. When operating in 22-bit addressing mode, the CPU memory management unit converts 16-bit virtual addresses into 22-bit real addresses. From the CPU side, this 2048K word real address space is divided as follows:

00000000 through 16777777 (0 through 1920K) access the main memory system,

17000000 through 17777777 (1920K through 2048K) access the UNIBUS.

The UNIBUS I/O page is at 17760000 through 17777777 (2044K through 2048K). Since there is no memory on the UNIBUS, the UNIBUS addresses below the I/O page are not useful to the CPU, except for maintenance.

The UNIBUS has 18 bits of address. To accommodate DMA transfers from and to devices on the UNIBUS, the PDP-11/44 and PDP-11/70 have separate pathways to main memory through a mapping box. The mapping box converts 18-bit addresses required by the device into 22-bit addresses required by the memory system. The mapping box consists of 32 registers, each containing a 22-bit base address. Thirty-one registers are available for use by device handlers.

When an 18-bit address is received from the UNIBUS side, the mapping box performs the following:

1 Selects one of the 31 registers using the five high-order (most significant) bits.

2 Adds the 13-low order (least significant) bits to the 22-bit base address to determine the 22-bit main memory address.

A 30-second mapping register is never used because the uppermost 4K words of address space on the UNIBUS are reserved for device registers (the I/O page). In effect, the mapping box takes the place of the whole 124K word memory system addressable on the UNIBUS.

On PDP-11/70s, transfers from and to MASSBUS devices occur through a third path that has full 22-bit addresses generated by the device controller (RH70). Refer to the *PDP-11 Processor Handbook* for details.

Transfers from and to UNIBUS DMA devices must use UNIBUS mapping registers (UMRs). A contiguous transfer of up to 4K words can be handled by each UMR.

Handlers can use UMRs for transfers in various ways depending on the nature of the device to be handled and the services provided by the handler. For example, a device supporting only one transfer at a time can preallocate UMRs to map up to the maximum transfer size supported for the device; for example, four UMRs for up to 16K words of transfer.

If a handler supports multiple DMA transfers at a time, it can allocate UMRs dynamically on a per transfer basis and deallocate them upon I/O completion. The handler must be able to cope with the event of no UMRs being available for a transfer; it must reissue allocation requests as UMRs become available.

A maximum of 31 independent transfers can be supported simultaneously by the UNIBUS mapping box.

To support more than 31 transfers, buffer pooling is necessary. A number of contiguous UMRs can be mapped into a buffer pool, for example, the SCOM node pool. Transfers can be made into and out of buffers in the pool. In this way, the number of simultaneous transfers supported by UMRs is virtually unlimited.

A UMR can map only on an even UNIBUS address. Handlers for devices, such as the DH11, that permit DMA transfers starting at an odd boundary must supply an offset from the UNIBUS address of the corresponding UMR in addition to filling the UMR.

For example, if the real buffer address is 364201 and UMR3 is being used, the UMR could be set to 364200 and the UNIBUS address supplied to the device would be 60001.

8.1.1 Summary of Introduction

- To effect DMA transfers over the UNIBUS (18-bit bus) into and out of 22-bit address main memory, UNIBUS mapping registers (UMRs) must be used. The 18-bit address is interpreted as follows:
  - The high-order five bits contain the UMR number,
  - The low-order 13 bits are added to the 22-bit address in the corresponding UMR to provide the actual 22-bit address of the transfer.
- Each UMR can map up to 4K words. For transfers greater than 4K, contiguous UMRs are required. This is the same principle as for the KT11 except that DMA transfers normally step linearly through the address space.
- The device handlers may need to use UMRs in a variety of ways depending on the devices concerned and the level of support desired.

The operating system provides a set of routines to enable handlers to support DMA transfers conveniently. Refer to Section 8.3.

8.2 UMR Support Database

8.2.1 Allocation Bitmap (.UMRBM)

The SCOM communication region contains a two-word bitmap. If a bit is set in the bitmap, the corresponding UMR has been allocated for a handler or transfer. The correspondence between a single bit and a UMR is as follows:

1 Bit 0 of the first word (.UMRBM) corresponds to UMR 0.

2 Bit 0 of the second word (.UMRBM+2) corresponds to UMR 16.

3 Bit 15 of .UMRBM+2 corresponds to UMR 31.

The bitmap indicates allocation; it does not indicate that a transfer is in progress using the corresponding UMR.

8.2.2 Free UMRs

To indicate that a UMR is free (that is, no transfer is currently in progress), the UMR concerned is filled with the following 22-bit value:

- Word 0 contains zeros,
- Word 1 contains 74 (octal)

This value points to the start of the top 124K of the main memory. This memory is not accessible using UMRs for a DMA transfer. Attempts to gain access to this memory always cause a nonexistent memory trap or controller error.

8.2.3 Machine Indicator Word (.UMR22)

In the SCOM communication region, a word exists for use by handlers to determine whether the machine where the handlers are running requires UMR handling. The low byte of .UMR22 holds the bits set at system generation and the high byte (.UMR22+1) holds bits set by the power-up code and SAVE. The high byte is used by the handler at run time to determine whether the mapping box is switched on. The following is an example.

```
BITB    #ON.UM,.UMR22+1
BEQ     (nonUMR code)
(UMR code)
```

The bit ON.UM is set only if the system is running on a PDP-11/44 or PDP-11/70 that has more than 124K words of memory. ON.UM is defined in EXEC.STB.

Other bits used in this byte are listed below.

ON.22—CPU is in 22-bit mode.
ON.70—CPU is a PDP-11/70.
ON.44—CPU is a PDP 11/44.

Bit ON.UM also implies ON.22; therefore, handlers for UNIBUS devices need test only ON.UM. Handlers for MASSBUS devices which may be connected to the UNIBUS of a PDP-11/44 need to test both ON.44 and ON.22.

8.3 Handler Library Routines for UMR Support

IAS provides a generalized set of routines in the handler library HNDLIB that enable a handler task to allocate, fill, reset, and deallocate UMRs. These routines are flexible and can be used in many combinations to provide optimum performance for a specific application.

Although these routines can be called at various points in the handler code, they provide certain functions that must be performed by all PDP-11/44 and PDP-11/70 DMA device handlers at some point:

1 Determine whether UMRs are needed and, if so, allocate them,

2 Find UMRs for each read or write when initiated,

3 Convert the allocated UMR slot number into an 18-bit address for the device,

4 Fill the allocated UMRs with the correct address at transfer time,

5 Free the UMRs when no longer needed,

6 Deallocate UMRs upon exiting.

All UMR support routines described in this section use R3 to contain the slot/length word. The slot/length word contains the starting UMR number in the low-order byte and the number of UMRs allocated in the high-order byte.

8.3.1 UMR Allocation Routines

8.3.1.1 ..URAL (UMR Allocator)

The routine ..URAL allocates UMRs for a handler. It finds consecutive bits in the .UMRBM bitmap and sets them. It then returns the starting UMR number and length (that is, the slot/length word).

R3 must contain the number of UMRs to be allocated before ..URAL is called.

If the C-bit is set upon return, ..URAL failed to allocate the UMRs. None have been allocated. The handler may want to try again for fewer UMRs by recalling ..URAL with a smaller value in R3.

If the C-bit is clear, R3 contains the slot/length word.

8.3.1.2 ..ALMR (UMR Allocator)

The routine ..ALMR allocates UMRs for a specific data transfer. It is called by the handler immediately before it initiates the transfer.

R1 must contain the request node address for the transfer. ..ALMR expects:

R.PB - High 6 bits of real address in bits 4 to 9.
R.PB+2 - Low 16 bits of real address.
R.PB+4 - Transfer size (bytes).

R2 must contain the map register block address.

The map register block (MRB) is a six word block that must exist in each handler that calls ..ALMR and ..DEMR. The MRB contains information about preallocated UMRs and UMRs which are dynamically allocated for a particular data transfer. The MRB contains the following six words:

M.RN - REQUEST NODE ADDRESS OF UMR OWNER

M.PW - PRE-ALLOCATED SLOT/LENGTH WORD
M.DF - NUMBER OF PRE-ALLOCATED UMRS (CAN BE 0)
M.SL - SLOT/LENGTH WORD (SET BY ..ALMR)
M.UL - LOW 16 BITS OF UNIBUS ADDRESS (SET BY ..ALMR)
M.UH - HIGH 2 BITS OF UNIBUS ADDRESS (SET BY ..ALMR)

If UMRs are already preallocated (in M.PW) and the transfer is less than the preallocated space, ..ALMR uses these UMRs. Otherwise it attempts to allocate the required number of UMRs (using ..URAL; see Section 8.3.1.2). If this fails ..ALMR releases the preallocated UMRs (using ..URDA; see Section 8.3.2.1) and attempts again to allocate the required number of UMRs.

If the C-bit is set on return, ..ALMR failed to allocate the UMRs.

If the C-bit is clear, UMRs have been allocated successfully. In this case ..ALMR returns the following:

M.SL - Slot/length for the allocated UMRs.
M.UH - High 2 bits of real address in bits 4 and 5.
M.UL - Low 16 bits of real address.
R.PB - same as M.UH
R.PB+2 - same as M.UL

UMRs allocated by ..ALMR should be deallocated using ..DEMR (see Section 8.3.2.2).

NOTE:

1 **All UNIBUS handlers that do DMA transfers should call these routines. If UMRs are not required (because of the configuration on which the handler is running) ..ALMR will return with condition code C clear. In this way the handler will run on any configuration.**

2 **MASSBUS handlers should only call these routines if ON.44 is set in .UMR22+1.**

8.3.2 UMR Deallocation Routines

8.3.2.1 ..URDA (UMR Deallocator)

The routine ..URDA resets the bits in the .UMRBM bitmap.

R3 must contain the slot/length word of the UMRs to be deallocated before ..URDA is called.

No values are returned by ..URDA.

8.3.2.2 ..DEMR (UMR Deallocator)

The routine ..DEMR deallocates UMRs which were allocated using routine ..ALMR (see Section 8.3.1.2). It is called when the data transfer has completed.

R2 must contain the map register block address (see Section 8.3.2.1).

If there were pre-allocated UMRs before ..ALMR was called then ..DEMR ensures that these remain allocated. It deallocates any additional UMRs (using ..URDA; see Section 8.3.2.1).

On return from ..DEMR, M.SL is cleared and M.PW contains the slot/length of the preallocated UMRs (if any).

8.3.3 ..URFL (Provides 22-Bit Address for Transfer)

The routine ..URFL takes a request node and the slot/length word and performs the following:

- Converts the virtual address in R.PB(R1) to a real 22-bit address.
- Fills the UMRs defined by R3 with the appropriate 22-bit addresses for the transfer.

R1 must contain the address of the I/O request node and R3 must contain the slot/length word before ..URFL is called.

No values are returned by ..URFL.

8.3.4 ..URF2 (Provides 22-Bit Address for Transfer)

This routine takes a 22-bit address in a double word and fills UMRs defined by the slot/length word to map the transfer.

R3 must contain the slot/length word. R4 and R5 contain a 22-bit, double word real address.

For the 22-bit address, R4 uses bits 0 through 5 to contain the high six bits of the 22-bit address. When ..VXFR and ..VXUR (refer to Section 8.5) are called, the real address high-order six bits are shifted to the left by four bits (bits 4 through 9) in order to maintain compatibility with previous versions of RSX-11D and IAS.

Hence, if the handler intends to call ..URF2 with the R4 and R5 contents returned from ..VXUR, R4 must be shifted to the right by 4 bits as follows:

```
ASH    #-4,R4
```

The shift must occur before calling ..URF2.

No values are returned by ..URF2.

8.3.5 ..URFR (Frees UMRs)

The routine ..URFR sets the UMRs defined in the slot/length word to the value of 17000000 to indicate that they are not in use.

R3 must contain the slot/length word before calling ..URFR. No values are returned.

8.3.6 ..URAD (Converts Slot/Length To 18-Bit Address)

The routine ..URAD converts the slot/length word contained in R3 to an 18-bit transfer address. This is the UNIBUS address that accesses the UMR slot.

..URAD returns the following values:

- R3 remains unchanged
- R4 contains the high-order 2 bits of the 18-bit address in bits 4 and 5.
- R5 contains the low-order 16 bits of the 18-bit address.

8.3.7 ..URFN (Finds Free UMRs within Allocated Range)

The routine ..URFN finds a number of contiguous free UMRs among those already allocated to map a transfer, if possible. The transfer is defined by the virtual address of the I/O request node in R.PB and the length in bytes in R.PB+2.

Before calling ..URFN, R1 must contain the address of the I/O request node and R3 must contain the slot/length word defining the range of UMRs to be searched.

.URFN returns the following values:

If the C-bit is set, ..URFN could not find the slot and the contents of R3 are undefined.

If the C-bit is clear, R3 contains the slot/length word of the slot found for the transfer. The UMRs have been filled with the 22-bit addresses for the transfer.

8.3.8 ..REAL (Calculates Real Address)

The routine ..REAL computes a real 22-bit address:

R1 must contain the address of the I/O request node.
R2 must contain the virtual address of the task.

R4 must contain the ATL (active task list) node address for the task before calling ..REAL.

..REAL returns the following values:

R4 contains the high-order six bits of the 22-bit address in bits 0 through 5.
R5 contains the low-order 16 bits of the 22-bit address.

..REAL cannot be used to calculate the 22-bit real address of one of the handlers own virtual addresses in the range 60000-77777.

8.4 SCOM Buffers and UMR Transfers

If UMRs are in operation, the high-order UMRs (for example, 30, 29, and 28) are preallocated and mapped into SCOM. The appropriate bits are allocated in the .UMRBM bitmap.

Sometimes transfers are done directly into/out of buffers picked from SCOM. To convert the 16-bit virtual address of the buffer into an 18-bit UMR UNIBUS address, set the two high-order bits of the 18-bit address and use the 16-bit virtual address for the rest.

8.5 Verify Transfer (..VXFR AND ..VXUR)

The routines ...VXFR and ...VXUR verify that the address for the requested transfer is legal and return the 22-bit real address in R4 and R5 as for ...REAL.

- R1 must contain the address of the I/O request node
- R2 must contain the starting address of the transfer
- R3 must contain the transfer length
- R5 must contain the transfer direction (0 = write into buffer, 1 = read from buffer).

8.6 Fixed and Dynamic UMR Handling

A handler task can view UMR allocation as either a fixed or dynamic process relative to a particular transfer. The following are examples of how a handler might use the available routines to correctly map transfers.

8.6.1 Fixed UMR Handling

When fixed UMR handling is used, the device handler task allocates a number of UMRs for its transfers at load time and keeps them until it exits. For the simple handler that dequeues one request at a time, fixed handling can be the more efficient approach. Fixed handling requires calling UMR routines in the order described below.

After declaring itself resident, the handler determines whether it is running on a PDP-11/44 or PDP-11/70 with extended memory enabled, as described in Section 8.2.3. If it is, it allocates a number of UMRs. The number allocated is the number required for the maximum transfer size for that device. One UMR is needed for each 4K of transfer length.

Since every transfer uses the same UMRs, starting with the first (lowest numbered) UMR, the slot number could be converted into the UNIBUS address at this point and then saved. The following is an example of the code required:

```
CALL     ..DSUT          ;DECLARE RESIDENT
BCS      exit           ;FAILED - EXIT
BITB      #ON.UM,.UMR22+1    ;11/70 - NEED UMRS
BEQ      NOUMR          ;NO UMRS NEEDED
MOV      #8.,R3         ;GET 8UMRS, 32K MAX TRANSFER
CALL      ..URAL
BCS      exit1          ;FAILED - EXIT
CALL      ..URAD         ;GET 18-BIT ADDRESS
MOV      R4,XRFADD       ;SAVE IT
MOV      R5,XRFADD+2
MOV      R3,SLOTSV       ;SAVE SLOT
NOUMR:
.
.                set up parameters
.
CALL     ..CINT
```

If desired, the handler could take other actions rather than failing when the required number of UMRs is not available. It could keep calling ..URAL until it gets as many UMRs as are available; then at I/O time, it could fail transfers too large for the number of UMRs obtained. The handler would exit only if no UMRs are available.

Since the handler is always starting a transfer at the first UMR allocated to it, it would never have to call ..URFN. The find operation becomes unnecessary.

At the verification of read and write operations only, the handler calls ..VXUR instead of ..VXFR to validate UMR requests. Since the handler has the 18-bit UNIBUS address at that point and that transfer is to be initiated, immediately, the handler can then call ..URF2 instead of ..URFL to fill the UMRs. This approach saves the time required to recompute the 22-bit address. The following is an example of the code required:

```
        BITB   #ON.UM,.UMR22+1 ;UMRs NEEDED
        BEQ    NOUM        ;NO
        CALL   ..VXUR       ;VERIFY TRANSFER
        BCS    ERR         ;ERROR - NO GOOD
        MOV    SLOTSV,R3     ;GET SLOT R1, R5 ALL SET
        ASH    #-4,R4       ;SHIFT R4 BITS 4, 5 TO 0,1
        CALL   ..URF2       ;
        BR    OK          ;
NOUM:   CALL   ..VXFR       ;REGULAR VERIFY
        BCS    ERR
```

When the transfer is initiated, the handler puts the 18-bit UNIBUS address that was saved during initialization of the handler into the device bus address registers. This is done instead of putting the output from ..VXFR (R4 and R5) into the device bus address register.

Since ..URFN was never called, free UMR (..URFR) is not needed at ..IODN time. In the handler exit code, immediately after disconnecting from the interrupt, the handler deallocates the UMRs. The following is an example:

```
        CALL   ..DINT       ;DISCONNECT
        MOV    SLOTSV,R3     ;SLOT NUMBER TO R3
        CALL   ..URDA       ;DEALLOCATE UMRS
        CALL   ..DNRC       ;DECLARE NONRESIDENT
```

At power-up, the same sequence executed at handler initialization should be performed in the event that the system is saved with the handler loaded on one machine and booted on another type.

8.6.2 Dynamic UMR Handling

Dynamic allocation can include two types of UMR use: semi-dynamic where the UMRs are allocated at handler initialization but found on a per transfer basis, and dynamic where UMRs are allocated on a per transfer basis.

8.6.2.1 Semi-dynamic Handling

When semi-dynamic handling is used, the handler task calls ..URAL to allocate the necessary UMRs immediately after calling ..DSUT. This process occurs in the handler initialization code or at power-up time.

At transfer initiation, the handler would call ..URFN to find free UMRs within allocated range for that transfer. If the handler is unable to obtain the needed UMRs, it can either fail the transfer or put the request in an internal retry queue. If UMRs are obtained, the handler calls ..URAD to convert the slot/length word to an 18-bit UNIBUS address.

In the exit code, the handler could call ..URDA to deallocate the UMRs.

8.6.2.2 Totally Dynamic UMR Handling

When totally dynamic handling is to be used, the handler first verifies the transfer by calling ..VXUR. The 22-bit real address is obtained from this call. Then, allocation of the required number of UMRs is requested by calling ..ALMR. If they are not available, the transfer can be failed, or it can be placed in an internal retry queue.

After obtaining UMRs, the handler places the UNIBUS address into the device bus address register, and starts the transfer.

It is not necessary to find free UMRs within an allocated range (using ..URFN) because the allocation is only temporary.

At I/O completion, the handler deallocates the UMRs by calling ..DEMR.

At handler exit time, no special UMR action is required.

A System Subroutines

The system subroutines available to executive privileged tasks in the host operating system are grouped here according to their function. The subroutines are explained on the following pages or in Chapter 8.

Routines that are in HNDLIB (see Section 6.2) are marked H. Routines that are in SCOM are marked S.

1. Interrupt Handling

..CINT (A.1.1) H
..DINT (A.1.2) H

2. Declaring Residency/Nonresidency

..DSUT (A.2.1) H
..DSMU (A.2.2) H
..DNRC (A.2.3) H

3. I/O Completion

..IODN (A.3.1) H

4. I/O Request Handling

..DQRE (A.4.1) H
..DQRN (A.4.2) H
..DISP (A.4.3) H
..VACC (A.4.4) H

5. Node Handling

..PENP (A.5.1) S
..PICK (A.5.2) S
..NADD (A.5.3) S
..NDEL (A.5.4) S
..IPRI (A.5.5) S
..RNTP (A.5.6) S
..PENV (A.5.7) S
..PICV (A.5.8) S
..RNTV (A.5.9) S
..NADV (A.5.10) S

6. Setting/Clearing Event Flags

..SEFN (A.6.1) S
..CEFN (A.6.2) S
..STEF (A.6.3) S
..CLEF (A.6.4) S

7. Attaching/Detaching a Unit

..ATUN (A.7.1) H
..DTUN (A.7.2) H

8. I/O Rundown/Kill All Requests

..FLSH (A.8.1) H
..FIFL (A.8.2) H

9. Information Transferring

..VXFR (A.9.1) H
..BLXO (A.9.2) H
..BLXI (A.9.2) H
..VXUR (ch. 8) H
..REAL (ch. 8) H

10. Swapping Page Descriptors

..SPD3 (A.10.1) S
..SPD4 (A.10.2) S
..SPD5 (A.10.3) S

11. Task Switching

..ENBO (A.11.1) S

12. Error Logging

..ERLI (A.12.1) S
..ERLD (A.12.2) H

13. IAS Task Swapping

..FRSW (A.13.1) H
..TKBK (A.13.2) H

14. UMR Handling

Refer to ch. 8. H

15. Power Fail Recovery

..PWUP (A.15.1) H

A.1 Interrupt Handling

A.1.1 ..CINT

The ..CINT subroutine is called to connect to an interrupt vector. This subroutine gets a node from the pool and charges it to the handler task. The node is to be used for the interrupt service routine. The format of the node ISR is detailed in Section B.5.

The ..CINT subroutine checks the interrupt vector and sets the interrupt trap vector to point to the interrupt node if no other user is attached to the interrupt vector.

If an interrupt service routine is not resident in the same contiguous area of memory as the device handler task (for example, if the interrupt service routine is in an SGA), then calling ..CINT fails.

The following registers must be preset before calling ..CINT:

- R0 must contain the interrupt vector address.
- R1 must contain the entry point of the interrupt service routine.
- R2 must contain the base address of the interrupt service space.
- R3 must contain the status of the condition codes (C,V,Z,N) upon entry to the interrupt service routine in bits 0 through 3 and the priority of the interrupt service routine in bits 5 through 7.

If ..CINT is not successful, the C condition code bit is set. This indicates that some other task has already been connected to the interrupt.

The exit condition for ..CINT is as follows:

- Virtual address 0 of the handler task contains the address of the interrupt code node.
- There are two global offsets from that address:
  1. I.MK - the offset of the bit mask
  2. I.MD - the offset of the I/O bus activity bit map.

Saving the contents of the two offsets enables you to set the I/O bus activity bit for the corresponding device by doing a bit-set of the mask into the location word.

A.1.2 ..DINT

The ..DINT subroutine is called to disconnect from an interrupt vector. This subroutine performs no checking. It sets the interrupt vector to the nonexistent interrupt status and returns the interrupt service routine node to the pool, thereby disconnecting the handler task from the interrupt vector. If the issuing task is not the task that is connected to the interrupt vector, the request is ignored. ..DINT must not be called unless the interrupt vector has been connected using ..CINT.

R0 must be preset to contain the interrupt vector address before calling ..DINT.

A.2 Declaring Residency/Nonresidency

A.2.1 ..DSUT

The ..DSUT subroutine is called by a non multiuser handler task to declare itself resident and to set values in the UIT. This subroutine establishes pointers in the UIT to each unit's respective PUD entry. The UIT is not necessarily ordered by unit number. Before entering ..DSUT, the unit number(s) of the desired unit(s) must be in UIT word(s) A. If no entries for the specified PUD are located, an error condition is returned. If entries are found, their count is returned.

For each PUD entry that has a handler successfully declared resident, ..DSUT fills PUD slot U.SL with the virtual address (handler virtual address) of the UIT entry concerned.

The UIT is detailed in Chapter 2.

The following registers must be preset before calling ..DSUT:

- R0 must contain the address of the UIT.
- R2 must contain the device type (2 ASCII characters).
- R3 must contain the flag byte for the PUD (all units).

After execution of ..DSUT, R1 contains the number of units found. If ..DSUT is not successful, the C condition code is set.

A.2.2 ..DSMU

The ..DSMU subroutine is called by a multiuser handler task (see Section 1.7) to declare itself resident and to set values in the UIT (see Table 2–1). Before entering ..DSMU the first or only word A of the UIT must contain the unit validity mask, described in Section 2.1.

The following registers must be preset before calling ..DSMU:

- R0 must contain the address of the UIT.
- R2 must contain the device type (2 ASCII characters).
- R3 must contain the flag byte for the PUD (all units).

..DSMU checks whether the device name in the handler's TI assignment matches the device type declared in R2, and whether the unit validity mask shows that the handler is being run for a legal unit.

..DSMU then calls ..DSUT (see Section A.2.1) to set values in the UIT and to fill (each) PUD slot U.SL with the handler virtual address of the UIT entry concerned.

After successful execution of ..DSMU with ..DSUT, R1 contains the number of units found. If ..DSMU or ..DSMU with ..DSUT is not successful, the C condition code is set.

A.2.3 ..DNRC

The ..DNRC subroutine is called to clear PUD entries (Word A) and declare the handler not resident. In addition, it clears the attach flag (Word 1, byte 1) in the UIT, and removes all nodes from the I/O request queue that are queued to the device handler header. The nodes are returned to the pool after performing an ..IODN.

R0 must be preset to contain the address of the UIT.

A.3 I/O COMPLETION

A.3.1 ..IODN

The ..IODN subroutine is called to complete the user's I/O request. This subroutine queues an I/O completion event to the task. This causes the I/O status block to be transferred to the user's area and an AST (if required) to be queued the next time the system context switches to the user's task. ..IODN decrements the I/O in-progress count and the transfers-pending count, and sets an event flag (if specified).

If an error occurs and the optional error logging subroutine is active, the error is logged, then remaining error logging procedures will be processed.

..IODN also checks the node to be returned to determine whether it is the I/O RUNDOWN node. If it is, ..IODN changes the status of the user's request to I/O RUNDOWN in progress (TS.IR3) and declares a significant event.

The following registers must be preset before calling ..IODN:

- R1 must contain the address of the request node.
- R2 must contain the adjustment to the decrement for the I/O in progress count.
- R3 must contain the I/O status block word 0 (WD.00).
- R4 must contain the I/O status block word 1 (WD.01).

A.4 I/O Request Handling

A.4.1 ..DQRE

The ..DQRE subroutine is called to dequeue an express request node from the handler's I/O request queue.

NOTES:

1 **Express request nodes can be identified since they have bit RF.XR set in offset location R.FC in the I/O request node.**

2 **The handler's local event flag 2 is set when an express request is queued.**

..DQRE is not affected by attached units. It always takes the express node at the top of the queue (that is, with the highest priority) and passes it to the handler task. ..DQRE searches for an express node by starting with the first entry in the UIT and scanning until it finds a node to dequeue or wraps around the UIT.

..DQRE does not attempt to dequeue a node from the queue if the express request node address (word C) in the UIT entry is nonzero.

Task switching is inhibited during the scan of the queue to prevent ..IPRI from inserting a node into the queue while one is being removed from it.

..DQRE increments the requests-in-progress count for a task to prevent checkpointing while an I/O operation is in progress.

The following exit conditions are established by ..DQRE:

- R0 contains the address of the UIT.
- R1 contains the address of the request node or is undefined if no node is located.

- R2 contains the address of the PUD pointer in the UIT unit entry or is zero if no node is located.

The C condition bit is set if no node is located.

Event flag 2 is cleared if no node is found; otherwise it remains set.

A.4.2 ..DQRN

The ..DQRN subroutine is called to dequeue a normal request node from the handler's I/O request queue.

NOTE: The handler's local event flag 1 is set when a normal request is queued.

..DQRN determines whether a task has attached the PUD of the requested unit. If a task has not attached the PUD, it takes the node from the top of the queue (i.e., with the highest priority) and passes it to the handler task. If a task has attached the PUD, ..DQRN searches the queue to find a node from the attaching task. If it cannot locate a node for that task, ..DQRN returns with condition codes set to indicate failure to dequeue a node.

..DQRN does not attempt to dequeue a node if the normal request node address in word B of the UIT entry is nonzero.

Task switching is inhibited during the scan of the queue to prevent ..IPRI from inserting a node into the list while one is being removed.

..DQRN increments the requests-in-progress count for a task to prevent checkpointing while an I/O operation into a user's area is taking place. The increment is not performed for a function that does not read or write into user space.

R0 must be preset to the address of the UIT before calling ..DQRN.

The following exit conditions are established by ..DQRN:

- R0 contains the address of the UIT.
- R1 contains the address of the request node or is undefined if no node is found.
- R2 contains a pointer to the PUD pointer in UIT word(s) A.

The C condition code is set if no node is found; otherwise, it is clear.

Event flag 1 is cleared if no node is found; otherwise, it remains set.

A.4.3 ..DISP

The ..DISP routine is entered to dispatch an I/O request according to the specified function code. The I/O function code is used to index into the dispatch table. The dispatch point within the handler or the ACP task is determined and the appropriate path taken.

The following registers must be preset before calling ..DISP:

- R0 must contain the address of the UIT.
- R1 must contain the request node address.
- R2 must contain the address of the PUD pointer in the UIT.

R0 acts as a pointer to the dispatch table's address which is held in the first location of the UIT.

If the function is an ACP function ..DISP does the following:

- Checks that the device is mountable; if not is returns an error.
- Builds a three-word data block:
  - Request node address
  - Pointer to PUD for device
  - I/O function code
- Sends the data block to the ACP using a Send and Request directive (VSDR$).

..DISP establishes the following exit conditions:

1 The C condition code is set to indicate an error in the SEND directive. ..DISP returns at the address in the first word of the dispatch table.

2 The C condition code is cleared to indicate a normal return. If the SEND directive was issued, ..DISP returns at the address in the second word of the dispatch table; otherwise, ..DISP returns at the address in the dispatch table for the I/O function code.

The ..DISP routine assumes that the I/O function codes will map into the dispatch table properly. Therefore, the handler needs to "pre-check" the I/O function codes.

A.4.4 ..VACC

The ..VACC subroutine is called to validate an I/O request. The I/O function code is used to index into the dispatch table. The access control information is validated against the access control information in the volume control block.

The following registers must be preset before calling ..VACC:

- R0 must contain the address of the start of the UIT.
- R1 must contain the request node address.
- R2 must contain the address of the PUD pointer in the UIT.

If the I/O request is rejected, the C condition code is set. If the request is valid, the C condition code is cleared.

A.5 Node Handling

A.5.1 ..PENP

The ..PENP subroutine is called to pick an empty 16-word node from the pool. This subroutine charges the requesting user with a node by incrementing the pool usage count by 2. ..PENP returns without a node if the specified user has exceeded his pool usage limit.

R1 must be preset to contain the system task directory (STD) address of the node user to be charged with the node.

If ..PENP is successful, R1 contains the address of the node.

If ..PENP is not successful, the C condition bit is set, indicating no node was picked.

A.5.2 ..PICK

The ..PICK subroutine is a general routine used to get a node from a deque (double-ended node queue). It ensures that a node exists in the deque, but does no accounting of the nodes. ..PICK inhibits interrupts to prevent a new node from being added to the deque while it gets a node.

R4 must be preset to contain the address of the deque head or the address of the previous node picked.

If ..PICK is successful, the C condition bit is cleared and R4 contains the address of the node.

If ..PICK is not successful, the C condition bit is set and/or R4 contains zero. The C condition bit can be set only if R4 was preset to the address of the deque head. If C = 1, no node was picked.

A.5.3 ..NADD

The ..NADD subroutine is called to add a node to a deque (double-ended node queue). The addition of the node is accomplished with interrupts inhibited, thus preventing a conflict with node deletion.

The following registers must be preset before calling ..NADD:

- R1 must contain the address of the node to be added.
- R4 must contain either the address of the start of the queue if the node is to be added to the front of the queue or the address of the previous node picked, if the node is to be added to the middle or end of the queue.

A.5.4 ..NDEL

The ..NDEL subroutine is called to delete a node from a queue. Because ..NDEL is a basic subroutine to delete a node from any list, it does not perform any accounting or check the node. It assumes that there is a node available and inhibits interrupts to prevent conflicts with nodes being added to a deque (double-ended queue).

R4 must be preset to contain a pointer to the node to be deleted before calling ..NDEL.

A.5.5 ..IPRI

The ..IPRI subroutine is called to insert a node in a queue according to its priority. It searches a queue (task switching is inhibited) and inserts a node in the correct position. If a node with the same priority as the node to be inserted is already in the list, the new node follows any others with the same priority; i.e., the order is first-in/first-out (FIFO). ..IPRI expects the node to have a priority in the common node priority position (R.PR).

The following registers must be preset before calling ..IPRI:

- R1 must contain the address of the node.
- R2 must contain the deque head.

..IPRI cannot be called from an ISR because it would lower processor priority to 3.

A.5.6 ..RNTP

The ..RNTP subroutine is called to return a 16-word node to the pool. It also decrements the pool usage count for the user specified in the common pool usage word by 2.

R1 must be preset to contain the address of the node to be returned before calling ..RNTP.

Word N.AW of the node must contain the address of the STD entry of the task to which the node is charged.

A.5.7 ..PENV

The ..PENV subroutine is called to pick an empty variable-sized node in 8-word blocks from the pool. The requesting user is charged for the node by adding the number of 8-word blocks allocated to his pool usage count. ..PENV returns without a node if the specified user has exceeded his pool usage limit or if the number of 8-word blocks requested is not available. The following registers must be preset before calling ..PENV:

- R1 must contain the user system task directory (STD) address to be charged with the node.
- R3 must contain the number of eight-word blocks requested.

If ..PENV is successful, R1 contains the address of the node. If ..PENV is not successful, the C condition code bit is set and R0 through R5 are unchanged.

Word N.AW of the node must contain the address of the STD entry of the task to which the node is charged.

A.5.8 ..PICV

The ..PICV subroutine is called to pick an empty variable-sized node in 8-word blocks from the pool without charging it to the requested user.

The following register must be preset before calling ..PICV:

R3 must contain the number of 8-word blocks requested.

If ..PICV is successful, R4 contains the address of the node that was picked and other registers are unchanged.

If ..PICV is not successful, the C condition bit is set and R4 will be undefined.

Neither ..PENV nor ..PICV should be called from an interrupt service routine and/or the processor priority should be no greater than 3 when calling ..PENV or ..PICV. If either condition is violated, the subroutine will be unsuccessful and the C condition code bit will be returned.

A.5.9 ..RNTV

The ..RNTV subroutine is called to return a variable-sized node to the pool and decrement the pool utilization count of the owner by the length of the node.

The following registers must be preset before calling ..RNTV:

- R1 must contain the address of the node to be returned.
- R3 must contain the length of the node in 8-word blocks.

Upon return, all registers will be unchanged.

Word N.AW of the node must contain the address of the STD entry of the task to which the node is changed.

A.5.10 ..NADV

The ..NADV subroutine is called to return a variable-sized node to the pool without node accounting.

The following registers must be preset before calling ..NADV:

- R1 must contain the address of the node to be returned.
- R3 must contain the length of the node to be returned in 8-word blocks.

Upon return, all registers will be unchanged.

It is the user's responsibility to maintain a count of the number of eight-word blocks so that the proper number can be returned.

The calling restriction for a 16-word node pick (..PENV or ..PICV) or return node (..RNTV or ..NADV) remain unchanged. A pick or return of 8 or more 16-word nodes should not be done at a processor priority greater than 2.

A.6 Setting/Clearing Event Flags

A.6.1 ..SEFN

The ..SEFN subroutine is called to set an event flag for a handler task. It is a general routine to set any event flag and can be used by a handler task at the interrupt level. The main use of the subroutine is to set the requester's event flag for I/O completion.

The following registers must be preset before calling ..SEFN:

- R0 must contain the number of the event flag to be set.
- R4 must contain the address of the active task list entry.

A.6.2 ..CEFN

The ..CEFN subroutine is called to clear an event flag for a handler task. It is a general routine to clear any event flag and can be used by a handler task at the interrupt level.

The following registers must be preset before calling ..CEFN:

- R0 must contain the number of the event flag to be cleared.
- R4 must contain the address of the active task list entry.

A.6.3 ..STEF

The ..STEF subroutine is called to set a device handler's event flags (1 through 16). The main function of the routine is to set event flags for the handler at interrupt level; however, it can also set event flags for any task if there is a PUD entry for the task. ..STEF sets any specified combination of the first 16 event flags; it does not affect any flags above 16. Once the flags are set, a significant event is declared.

The following registers must be preset before calling ..STEF:

- R0 must contain the PUD entry address.
- R1 must contain the flags' indicator,

  that is:
  bit 0 = event flag 1
  bit 1 = event flag 2 etc.

A.6.4 ..CLEF

The ..CLEF subroutine is called to clear a handler's event flags (1 through 16). Its main purpose is to clear an event flag of the handler task after an interrupt has been recognized. It can be used to clear more than one event flag.

The following registers must be preset before calling ..CLEF:

- R0 must contain the PUD entry address.
- R1 must contain the flags' mask,

  that is:
  bit 0 = event flag 1
  bit 1 = event flag 2

A.7 Attaching/Detaching a Unit

A.7.1 ..ATUN

The ..ATUN subroutine is called to attach a unit to a task. It ensures that a task is not already attached to the specified PUD entry and then attaches the task to the PUD. It increments the transfers pending count for the task to prevent it from exiting without detaching the device (PUD entry).

The following registers must be preset before calling ..ATUN:

- R1 must contain the request node address.
- R2 must contain a pointer to the PUD address. The pointer is obtained from the UIT.

A.7.2 ..DTUN

The ..DTUN subroutine is called to detach a task from a unit. It ensures that the task requesting detachment is the one attached to the unit and then detaches the task from the specified PUD entry. It then decrements the transfers pending count for the task and exits.

The following registers must be preset before calling ..DTUN:

- R1 must contain the request node address.
- R2 must contain a pointer to the PUD address. The pointer is obtained from the UIT.

If ..DTUN is not successful, the C condition code is set.

A.8 I/O Rundown and Kill All Requests

A.8.1 ..FLSH

The ..FLSH subroutine is called to remove all requests for the specified task from the queue and detach the unit for I/O RUNDOWN and KILL ALL REQUESTS. This subroutine decrements the transfers-queued and transfers-pending count for the task to allow it to exit and clears the attach bit in the PUD if it is set.

R1 must be preset to the address of the request node before calling ..FLSH.

In addition, the request node must contain the following information:

- Parameter 1 = ATL node,
- Parameter 2 = STD node, and
- Parameter 3 = PUD pointer.

A.8.2 ..FIFL

The ..FIFL subroutine is called to perform a SEND/REQUEST to the files system to deaccess files.

R1 must be preset to contain the I/O RUNDOWN or KILL ALL REQUESTS node address.

Condition codes are set upon exit according to the SEND/REQUEST EMT.

A.9 Information Transferring

A.9.1 ..VXFR

The ..VXFR subroutine is called to validate a user's transfer request. ..VXFR determines whether the user's request to transfer data into or out of his area is legal. Also, ..VXFR ensures that transfers across virtual bounds are also transfers in contiguous core. It does not allow the user to transfer into read-only space if the direction specified by the handler is write. After validation by ..VXFR, a handler is not concerned with the legality of transferring a contiguous block of core into the user's area. ..VXFR returns the physical 18-bit address of the user's buffer in R4 and R5.

The following registers must be preset before calling ..VXFR:

- R1 must contain the request node address.
- R2 must contain the virtual starting address. For an Executive request, R2 must contain the mod. 64. of a physical address.

- R3 must contain the transfer length.
- R5 must contain the direction of the transfer: 0 = I/O device to memory, 1 = memory to I/O device.

If ..VXFR executes without detecting an error, the following exit conditions are set:

- R4 contains the high-order address (bits 4 and 5).
- R5 contains the low-order address.

If ..VXFR detects an error, the C condition code bit is set.

Since the Executive does not issue a load record task QIO (it is done by the normal, logical read/write QIO), there is no requirement for those handlers having this capability, to issue a load record task QIO. Therefore, DO NOT check for an odd byte count BEFORE calling ..VXFR.

A.9.2 ..BLXO and ..BLXI

The ..BLXO or ..BLXI subroutine is called to pass information to or from a user's address space, respectively. ..BLXO validates and transfers the request to transfer out of a handler task area. ..BLXI validates and transfers the request to transfer into a handler task area. The information transferred should not be mapped via APR3. ..BLXO and ..BLXI use APR3 as a scratch area. (Compare Section 1.1.)

The following registers must be preset before calling ..BLXO or ..BLXI:

- R1 must contain the request node address.
- R2 must contain the virtual starting address.
- R3 must contain the transfer length in bytes.
- R4 must contain the handler virtual address.

If an error is detected, the C condition code is set.

Use of ..BLXO and ..BLXI is restricted to transfers with a maximum word length of 4032 (decimal).

A.10 Swapping Page Descriptors

The subroutines used for swapping page descriptors cannot be called by an ISR because an ISR does not have the normal handler task structure, and therefore has no APRs to use in swapping.

A.10.1 ..SPD3

The ..SPD3 subroutine is called to swap page descriptor 3 with two words on the stack. ..SPD3 gets two words from the stack and swaps them with the page descriptors on the stack.

The ..SPD3 routines are used to establish the page descriptor and address registers of a handler task. They allow a handler task to access areas other than its own and prevent the APR's from being modified during task switching.

The following calling sequence is required:

```
MOV  PAGE DESCRIPTOR,-(SP)
MOV  PAGE ADDRESS REGISTER,-(SP)

CALL ..SPD3
```

When ..SPD3 has executed, the stack contains the old page and address descriptors.

A.10.2 ..SPD4

The ..SPD4 subroutine is called to swap page descriptor 4 with two words from the stack. ..SDP4 performs the same functions for page descriptor 4 as ..SPD3 does for page descriptor 3. Likewise, the calling sequence and exit conditions are the same. Page descriptor 4 must be restored if it refers to system communication area.

A.10.3 ..SPD5

The ..SDP5 subroutine is called to swap page descriptor 5 with two words from the stack. ..SPD5 performs the same functions for page descriptor 5 as ..SPD3 does for page descriptor 3. Likewise, the calling sequence and exit conditions are the same. Page descriptor 5 must be restored if it refers to the system communication area.

A.11 TASK SWITCHING

A.11.1 ..ENB0

The ..ENB0 subroutine is called to enable task switching. ..ENB0 enables task switching and allows the system to update the clock and recognize significant events. When this routine is entered, the Processor Status Word must have been saved on the top of the stack. Normally, the .INH0 macro is used to set up for ..ENB0.

..ENB0 does not enable task switching if the priority saved on the stack does not indicate a priority of 0.

When the processor status word is saved by .INH0 it is always on the top of the stack.

The .INH0 macro is defined as follows:

```
.MACRO      .INH0
        MOV @#PS.EXP,-(SP)
        BISB #140,@#PS.EXP
.ENDM
```

A.12 Error Logging

Error logging is a routine that gathers information on device errors that occur during processing, and then produces a report from that information.

A.12.1 ..ERLI

The ..ERLI subroutine is called from the ISR when an error is detected.

The following registers must be preset before calling ..ERLI:

- R1 must contain the request node address.
- R3 must contain the number of device registers to be passed.
- R4 must contain the beginning address of the double-word device statistics block.

  This block must be addressable by Kernel APR3.

R5 must contain the maximum retry count (high byte) and the current retry count (low byte).

If ..ERLI is successful, R0 through R4 are unchanged, and R5 contains the starting address within the dynamic buffer which the handler will use while passing its device registers.

If ..ERLI is not successful, the C condition code bit is set on return to indicate one of the following error conditions:

1 An error was already logged for a QIO.

2 The error logging task is not active (may be checkpointed or not running).

3 Dynamic buffer is not available to log the error.

When ..ERLI is not successful, the contents of R5 are undefined.

It is the user's responsibility to fill in a byte that is offset from R5 as the unit descriptor word (UDW). The offset global symbol is .ELOF. The UDW is used by the error logging analyzer to identify the characteristics of the device.

A.12.2 ..ERLD

The ..ERLD subroutine is called by ..IODN when an error is detected.

The ..ERLD subroutine tries to complete the information required for error logging, but which is not needed at the ISR level. Once the information is obtained, ..ERLD passes the information to the error logging task (if that task is active).

If ..ERLI calls ..ERLD, then ..ERLD is "transparent" to the handler task.

The following register must be preset before calling ..ERLD:

R1 must contain the request node address.

A.13 IAS Task Swapping

The following routines are used by a handler to enable task swapping while an I/O operation is in progress and to lock a task back in memory.

A.13.1 ..FRSW

The .FRSW routine is called to free a task for swapping while a current I/O request is being serviced. Freeing a task means that it becomes elegible for swapping. The routine releases any memory locks for the requests and increments the count of swap I/O requests for the task.

The following register must be preset before calling ..FRSW:

R1 must contain the request node address.

A.13.2 ..TKBK

The ..TKBK routine is called to cause a task to be reloaded and locked in memory until either it is freed (via ..FRSW) or the I/O done routine is called (..IODN).

If the task is not in memory, the routine will mark the task to be reloaded and return with condition code C set. When the requested task is back in memory, the system will set event flag 3 for the handler. The handler must call this routine again after responding to event flag 3 to ensure that the task is still in memory.

If the task is in memory, the routine will reapply memory locks to prevent swapping and shuffling of the transfer area, decrement the swap I/O count for the task, and return with condition code C clear.

The following register must be preset before calling ..TKBK:

R1 must contain the request node address.

A.14 UMR Handling

See Chapter 8.

A.15 Power Fail Recovery

A.15.1 ..PWUP

The ..PWUP routine is called to wait for mounted units to become ready again after a power failure. ..PWUP requires an auxiliary routine as described in Section 4.7.

The following registers must be preset before calling ..PWUP:

- R0 must contain the timeout period in seconds.
- R1 must contain the address of the auxiliary routine.
- R2 must contain the address of the UIT.

B Symbolic Definitions

The format, content, and offsets for the physical unit directory, the system task directory, the active task list, clock nodes, and an I/O request node follow.

B.1 Physical Unit Directory (PUD)

The PUD is a fixed list of entries describing each physical device/unit in the system. The PUD is created during system generation and consists of entries with the following format:

```
.SBTTL PUD -- PHYSICAL UNIT DIRECTORY
;
; PUD -- PHYSICAL UNIT DIRECTORY
;
; THE "PUD" IS A FIXED LIST OF ENTRIES DESCRIBING EACH PHYSICAL DEVICE-
; UNIT IN A SYSTEM.  THIS LIST IS CREATED BY THE SYSTEM CONFIGURATION
; ROUTINE (SYSGEN) AND CONSISTS OF ENTRIES OF THE FOLLOWING FORMAT:
;
U.DN==00    ; WD. 00 (B 00) -- DEVICE NAME (2 ASCII CHARS)
U.UN==02    ; WD. 01 (B 02) -- UNIT NUMBER (BYTE)
U.FB==03    ;         (B 03) -- FLAGS (BYTE)
U.C1==04    ; WD. 02 (B 04) -- CHARACTERISTICS WORD ONE (DEVICE INDEPENDENT INDICATORS)
U.C2==06    ; WD. 03 (B 06) -- CHARACTERISTICS WORD TWO (DEVICE DEPENDENT INDICATORS)
U.C3==10    ; WD. 04 (B 10) -- CHARACTERISTICS WORD THREE (DEVICE DEPENDENT INDICATORS)
U.C4==12    ; WD. 05 (B 12) -- CHARACTERISTICS WORD FOUR (SIZE OF BLOCK, BUFFER, LINE)
U.AF==14    ; WD. 06 (B 14) -- ATTACH FLAG - EITHER:
    ;
    ;   1. ATL ADDRESS OF ATTACHED TASK, OR
    ;   2. PUD ADDRESS OF OWNING DEVICE (IAS
    ;      ONLY, IF UC.IAS OR UC.IEX SET)
    ;
U.RP==16    ; WD. 07 (B 16) -- REDIRECT POINTER
U.HA==20    ; WD. 10 (B 20) -- HANDLER TASK ATL NODE ADDRESS
U.XC==22    ; WD. 11 (B 22) -- COUNT OF EXPRESS REQUESTS IN QUEUE
U.RF==24    ; WD. 12 (B 24) -- UNIT REQUEST DEQUE LISTHEAD (FWD PNTR)[ OBSOLETE]
U.SL==24    ; WD. 12 (B 24) -- ADDRESS OF UIT SLOT FOR THIS DEVICE IN
    ;        HANDLER TASK
U.SC==26    ; WD. 13 (B 26) -- ADDRESS OF SCB(SHADOW CONTROL BLOCK) FOR DISK+
U.TV==30    ; WD. 14 (B 30) -- INTERRUPT TRAP VECTOR ADDRESS
U.IP==32    ; WD. 15 (B 32) -- INTERRUPT PRIORITY (IN BITS 5-7)
U.DA==34    ; WD. 16 (B 34) -- [DEVICE PAGE ADDRESS]
;
; PHYSICAL UNITS ARE CONSIDERED "VOLUMES" BY THE FILES SYSTEM, AND THE
; REMAINDER OF THE PUD ENTRY IS A "VOLUME CONTROL BLOCK".
;
; Note that the first word in the Volume Control Block has an alternate use
; for a terminal device.
;
U.VA==36    ; WD. 17 (B 36) -- ADDRESS OF VOLUME CONTROL BLOCK EXTENSION
U.LA==36    ; WD. 17 (B 36) -- Logical assignment listhead for terminals
U.UI==40    ; WD. 20 (B 40) -- USER IDENTIFICATION CODE (UIC)
U.PC==40    ;         (B 40) -- UIC PROGRAMMER CODE
U.GC==41    ;         (B 41) -- UIC GROUP CODE
U.VP==42    ; WD. 21 (B 42) -- VOLUME PROTECTION WORD
U.CH==42    ;         (B 42) -- CHARACTERISTICS FLAGS
    ;        (B 43) -- UNUSED+
```

```
U.AR==44    ; WD. 22 (B 44) -- ACCESS RIGHTS FLAGS WORD
U.DACP==46  ; WD. 23 (B 46) -- DEFAULT ACP NAME, RAD50 (FIRST WORD)
U.ACP==50   ; WD. 24 (B 50) -- EITHER:
    ;
    ;    1. STD ENTRY ADDRESS OF CURRENT ACP
    ;       (FILE STRUCTURED DEVICES)
    ;    2. CORRESPONDING UTN ADDRESS (TIMESHARING
    ;       TERMINALS)
    ;
U.TF==52    ; WD. 25 (B 52) -- TERMINAL FLAGS WORD
U.PR==52    ; WD. 25 (B 52) -- TERMINAL PRIVILEGE BYTE
U.FO==53    ;        (B 53) -- TERMINAL FORMS BYTE
U.LBH==54   ; WD. 26 (B 54) -- HIGH ORDER - TOTAL # OF BLKS FOR DEVICE
U.LBN==56   ; WD. 27 (B 56) -- LOW ORDER- TOTAL # OF BLOCKS FOR DEVICE
U.XPUD==60  ; WD. 30 (B 60) -- Virtual Address of PUD extension in device handler
;
; THIS WORD IS ONLY USED IN AN IAS SYSTEM
;
U.TS==62  ; WD. 31 (B 62) -- COUNT OF USERS OF THE VOLUME
U.MN==63  ;        (B 63) -- IAS FLAGS
;
U.SZ==64
;
; FLAGS BYTE BIT DEFINITIONS
;
UF.RH==200  ; SET WHEN HANDLER TASK IS DECLARED RESIDENT.
UF.TL==100  ; SET WHEN HANDLER TASK RECOGNIZES LOAD AND RECORD
UF.OFL==040 ; SET WHEN DEVICE IS OFFLINE
UF.CO==020  ; SET WHEN DIRECTED TO CO
UF.ACT==010 ; SET WHEN DEVICE IS ACTIVE
UF.SC==004  ; SET WHEN DISK IS THE SHADOW COPY+
UF.SD==002  ; SET WHEN DISK IS ONE OF THE SHADOW PAIR+
UF.LCK==001 ; Set to prevent all access to a device except by it's owner
  ;
;
; BIT DEFINITIONS FOR CHARACTERISTICS WORD ONE
;
UC.REC==000001 ;[00] SET IF RECORD ORIENTED DEVICE (VIZ., TT, LP, CR)
UC.CCL==000002 ;[01] SET IF CARRIAGE CONTROL DEVICE (VIZ., TT, LP)
UC.TTY==000004 ;[02] SET IF TTY DEVICE (VIZ., KSR, LA30)
UC.DIR==000010 ;[03] SET IF DEVICE IS A DIRECTORY DEVICE
UC.SDI==000020 ;[04] SET IF DEVICE IS A SINGLE DIRECTORY DEVICE
UC.SQD==000040 ;[05] SET IF DEVICE IS A SEQUENTIAL DEVICE
UC.IAS==000100 ;[06] SET IF AN INTERACTIVE IAS TERMINAL
UC.IEX==000200 ;[07] SET IF AN IAS EXCLUSIVE DEVICE
UC.INB==000400 ;+003 [08] SET IF THE DEVICE IS INTERMEDIATE BUFFERED
UC.SWL==001000 ;[09] SET IF THE DEVICE IS SOFTWARE WRITE LOCKED
UC.ISP==002000 ;[10] SET IF DEVICE IS INPUT SPOOLED
UC.OSP==004000 ;[11] SET IF DEVICE IS OUTPUT SPOOLED
UC.PSE==010000 ;[12] SET IF DEVICE IS PSEUDO DEVICE
UC.COM==020000 ;[13] SET IF DEVICE IS COMMUNICATIONS CHANNEL
UC.F11==040000 ;[14] SET IF DEVICE IS FILES-11
UC.MNT==100000 ;[15] SET IF DEVICE IS MOUNTABLE
;
; Bit definitions for characteristics word two (mass storage devices)
;
U2.WCK==000001 ;[00] SET IS READ-AFTER-WRITE CHECK REQUIRED
U2.SYD==000010 ;[03] Unit is system device
U2.MOH==000020 ;[04] SET IF DEVICE HAS MOVING HEADS
U2.RMV==000040 ;[05] SET IF DEVICE HAS REMOVABLE VOLUMES
U2.BAD==000100 ;[06] SET IF DEVICE HAS FACTORY-SUPPLIED BAD BLOCK INFO
U2.CLS==017400 ; Mask for device class code (5 bits)
U2.TYP==160000 ; Mask for device type within class code
```

```
U2.DEV==177400 ; Mask for device Class/Type bits
U2.DNS==U2.TYP ; Density bits mask for magtape devices
;
; Bits for use with the above mask
;
U2D.62==020000 ; [13] Tape drive can handle 6250 bpi
U2D.FX==040000 ; [14] Tape drive has only one density
U2D.16==100000 ; [15] Tape drive can handle 1600 bpi
;
; Supported tape density is coded as follows -
;
; Density supported   Bit(s) set
;
; 800 bpi only    U2D.FX
; 800 and 1600 bpi   U2D.16
; 1600 bpi only    U2D.16!U2D.FX
; 1600 and 6250 bpi   U2D.16!U2D.62
;
; Device Class/Type definitions
;
; For line printer the lower 2 bits are defined as follows -
;
U2.LC ==001  ; Line printer support lower case
U2.LS ==002  ; Line printer is an LS11
;
;       Device Class     Device Type
;
U2.NIL ==000  ;  00  0 Unknown Device
U2.RF1 ==001  ; 01  0 RF11 (1-8 platters)
U2.RF2 ==041  ; 01  1
U2.RF3 ==101  ; 01  2
U2.RF4 ==141  ; 01  3
U2.RF5 ==201  ; 01  4
U2.RF6 ==241  ; 01  5
U2.RF7 ==301  ; 01  6
U2.RF8 ==341  ; 01  7
;
U2.K5 ==002  ; 02  0 RK05
U2.K3 ==042  ; 02  1 RK03
U2.5F ==102  ; 02  2 RK05F
;
U2.P2 ==103  ; 03  2 RP02
U2.P3A ==203  ; 03  4 RP03A
U2.P3B ==303  ; 03  6 RP03B
;
U2.P4 ==004  ; 04  0 RP04
U2.P5 ==044  ; 04  1 RP05
U2.P6 ==104  ; 04  2 RP06
;
U2.S3 ==005  ; 05  0 RS03
U2.S4 ==045  ; 05  1 RS04
;
U2.K6 ==006  ; 06  0 RK06
U2.K7 ==046  ; 06  1 RK07
;
U2.X1 ==007  ; 07  0 RX01
;
U2.T56 ==050  ; 10  1 TU56 DECtape
;
U2.M2 ==011  ; 11  0 RM02
U2.M3 ==051  ; 11  1 RM03
U2.M5 ==111  ; 11  2 RM05
U2.M80 ==151  ; 11  3 RM80
```

```
U2.P7 ==211  ; 11  4 RP07
;
U2.L1 ==012  ; 12  0 RL01
U2.L2 ==052  ; 12  1 RL02
;
U2.T58 ==013  ; 13  0 TU58 DECtape II
;
U2.X2 ==014  ; 14  0 RX02
;
U2.A8 ==056  ; 16  1 RA80
U2.A82 ==116  ; 16  2 RA82
U2.A9 ==156  ; 16  3 RA90
U2.A6 ==216  ; 16  4 RA60
U2.A81 ==256  ; 16  5 RA81
U2.A70 ==316  ; 16  6 RA70
;
U2.F25 ==057  ; 17  1 RCF25
U2.C25 ==117  ; 17  2 RC25
;
U2.D31 ==060  ; 20  1 RD31
U2.D32 ==120  ; 20  2 RD32
U2.D33 ==220  ; 20  4 RD33
;
U2.X33 ==123  ; 23  3 RX33
U2.D54 ==163  ; 23  3 RD54
U2.D53 ==223  ; 23  4 RD53
U2.D52 ==263  ; 23  5 RD52
U2.D51 ==323  ; 23  6 RD51
U2.X50 ==363  ; 23  7 RX50
;
U2.T10 ==130  ; 30  2 TU/TE10
U2.T16 ==231  ; 31  4 TU/TE16,TU45,TU77
U2.T11 ==332  ; 32  6 TS11,TU80
;
U2.T81 ==273  ; 33  5 MU TU81
U2.T50 ==333  ; 33  6 MU TK50
U2.T70 ==373  ; 33  7 MU TK70
;
U2.LP ==036  ; 36  0 Generic line printer
U2.LS ==076  ; 36  1 Generic LS printer

UC.WCK==U2.WCK ;++008 SET IF A READ AFTER WRITE CHECK IS REQUIRED
;
; BIT DEFINITIONS FOR VOLUME CHARACTERISTICS BYTE U.CH
;
CH.OFF==200 ;VOLUME IS OFF-LINE
CH.FOR==100 ;VOLUME IS FOREIGN
CH.UNL==40 ;DISMOUNT PENDING
CH.NAT==20 ;ATTACH/DETACH NOT PERMITTED
CH.NDC==10 ;DEVICE CONTROL FUNCTIONS NOT PERMITTED
CH.LAB==1 ;VOLUME IS LABELED TAPE
;
;
; BIT DEFINITIONS FOR TERMINAL PRIVILEGE BYTE
;
UT.PR==1 ;SET IF TTY IS PRIVLEGED
UT.SL==2 ;SET IF TTY IS SLAVED
UT.LG==4 ;SET IF TERMINAL IS LOGGED ON
;
; IAS FLAG BYTE DEFINITITION
UM.PR==001 ;[01] SET IF MOUNT/DISMOUNT IN PROGRESS
UM.GB==002 ;[02] SET IF VOLUME GLOBALY MOUNTED   /IAS
UM.RT==004 ;[03] SET IF MOUNTED FOR REAL TIME
UM.TS==010 ;[04] SET IF MOUNTED FOR TIMESHARING
```

```
UM.MC==020 ;[05] SET IF MCR MOUNT
UM.RLT==200 ;[08] SET IF A TASK NEEDS TO BE RELOADED FOR THIS DEVICE
;
; Offsets in the extended PUD used by the MSCP handler
;
X.FLGS==00 ; Wd. 00 (B 00)  -- Extended PUD flags must be the first word always
X.MLUN==02 ; Wd. 01 (B 02)  -- Multiunit code
X.UNTI==04 ; Wd. 02 (B 04)  -- Unit identifier
    ; Wd. 03 (B 06)  -- X.UNTI+2 - Unit identifier (cont.)
    ; Wd. 04 (B 08.) -- X.UNTI+4 - Unit identifier (cont.)
    ; Wd. 05 (B 10.) -- X.UNTI+6 - Unit identifier (model and class)
X.SN  ==14 ; Wd. 06 (B 12.) -- Volume serial number (lo order)
    ; Wd. 07 (B 14.) -- X.SN+2 - Volume serial number (hi order)
X.TRCK==20 ; Wd. 10 (B 16.) -- Track size (LBNs per track)
X.GRP ==22 ; Wd. 11 (B 18.) -- Group size (Tracks per group)
X.CYL ==24 ; Wd. 12 (B 20.) -- Cylinder size (Groups per cylinder)
X.USVR==26 ; Wd. 13 (B 22.) -- Unit software version number
X.UHVR==27 ;      (B 23.) -- Unit hardware version number
X.RCTS==30 ; Wd. 14 (B 24.) -- Replacement Control Table size
X.RBNS==32 ; Wd. 15 (B 26.) -- Number of RBNs per track
X.RCTC==33 ;      (B 27.) -- Number of RCT copies
X.AVLH==34  ; Wd. 16 (B 28.) -- RNA listhead for available status (ST.AVL) I/O
;
X.SZ  ==40 ; Size of extended PUD entry
;
; The following bits are defined in the extended PUD flags word
;
XF.ONL==200   ; Indicates unit online in progress
XF.BBR==100   ; Host Bad Block Replacement is supported
XF.AVL==40   ; ST.AVL return status online in progress
XF.FMT==20   ; Disk is in the process of being formatted
```

B.2 System Task Directory (STD)

The STD is a memory-resident directory of all tasks that have been installed into the system. The directory consists of two parts:

1 A fixed-size area of one word for each task that could be installed at a given time, and

2 An STD entry for each task that is installed.

The fixed-size area is referred to as the alpha table area. It provides space for alphabetically-ordered contiguous list pointers to STD entries to facilitate a binary search for STD entries by task name.

Each STD entry has the following format:

```
.SBTTL STD-- SYSTEM TASK DIRECTORY
;
; STD-- SYSTEM TASK DIRECTORY
;
; THE SYSTEM TASK DIRECTORY IS A MEMORY RESIDENT DIRECTORY OF ALL TASKS
; WHICH HAVE BEEN INSTALLED INTO A SYSTEM.  THIS DIRECTORY CONSISTS OF TWO
; PARTS: (1) A FIXED SIZE AREA OF ONE WORD FOR EACH TASK THAT MAY
; BE INSTALLED AT ANY TIME, AND (2) AN STD ENTRY FOR EACH TASK THAT IS
; INSTALLED.  THE FIXED SIZED AREA IS  CALLED THE "ALPHA TABLE" AND
; PROVIDES SPACE FOR AN ALPHABETICALLY ORDERED CONTIGUOUS LIST OF POINTERS
; TO STD ENTRIES TO FACILITATE SEACH FOR STD ENTRY BY TASK NAME.
; EACH STD ENTRY IS OF THE FOLLOWING FORMAT:
;
S.TN==00 ; WD. 00 (B 00) -- TASK NAME (6 CHAR IN RADIX-50, 2 WORDS)
  ; WD. 01 (B 02) -- (SECOND HALF OF TASK NAME)
S.TD==04 ; WD. 02 (B 04) -- DEFAULT TASK PARTITION (TPD ADDRESS)
S.FW==06 ; WD. 03 (B 06) -- FLAGS WORD
S.DP==10 ; WD. 04 (B 10) -- DEFAULT PRIORITY (BYTE)
S.DI==11 ;        (B 11) -- SYSTEM DISK INDICATOR (BYTE)
S.LZ==12 ; WD. 05 (B 12) -- 1/64TH SIZE OF LOAD IMAGE
S.TZ==14 ; WD. 06 (B 16) -- 1/64TH MAX TASK SIZE
S.AV==16 ; WD. 07 (B 16) -- NUMBER OF ACTIVE VERSIONS OF TASK (BYTE)
S.PV==17 ;    (B 17) -- TASK POOL LIMIT PER VERSION (BYTE)
S.PU==20 ; WD. 10 (B 20) -- TASK POOL UTILIZATION
S.RF==22 ; WD. 11 (B 22) -- RECEIVE DEQUE LISTHEAD (FWD PNTR)
S.RB==24 ; WD. 12 (B 24) -- RECEIVE DEQUE LISTHEAD (BKG PNTR)
S.DL==26 ; WD. 13 (B 26) -- LOAD IMAGE FIRST BLOCK NUMBER (32-BITS)
  ; WD. 14 (B 30)    (SECOND HALF OF DISK ADDRESS)
S.PA==32 ; WD. 15 (B 32) -- GCD NODE ADDRESS FOR PURE AREA
;
;
; THE SYSTEM DISK INDICATOR SPECIFIES WHICH I/O REQUEST QUEUE IS
; TO RECEIVE A "LOAD TASK IMAGE" REQUEST, BY PROVIDING A "PUD ENTRY INDEX".
; E.G., A ZERO WOULD INDICATE THE REQUEST QUEUE FOR THE DEVICE-UNIT
; REPRESENTED BY THE FIRST (ENTRY ZERO) PUD ENTRY.
;
; FLAGS WORD BIT DEFINITIONS:
;
SF.MK==000001 ;++031 [00] USED BY SGN1 TO MARK STD ENTRIES
SF.FX==000002 ;[01] SET WHEN TASK IS FIXED IN MEMORY
SF.RM==000004 ;[02] SET WHEN STD IS TO BE REMOVED
SF.TD==000010 ;[03] SET WHEN TASK IS DISABLED
SF.BF==000020 ;[04] SET  WHEN A TASK IS BEING FIXED IN MEMORY
SF.XT==000040 ;[05] SET WHEN A TASK IS TO BE REMOVED ON EXIT
SF.MU==000100 ;[06] SET WHEN TASK IS MULTI-USER
SF.PT==000200 ;[07] SET WHEN TASK IS A PRIVILEGED TASK
SF.NT==000400   ;[08]  NETWORK ATTRIBUTE BIT
SF.R1==001000 ;[09] RESTRICTED USAGE LEVEL ONE (BACKGROUND BATCH JOBS)
SF.XS==002000 ;[10] TASK NOT ABLE TO RECEIVE DATA OR REFERENCES
SF.XA==004000 ;[11] SET WHEN TASK IS NEVER TO BE ABORTED
SF.XD==010000 ;[12] SET WHEN TASK IS NEVER TO BE DISABLED
SF.XF==020000 ;[13] SET WHEN TASK IS NEVER TO BE FIXED IN MEMORY
SF.XC==040000 ;[14] SET WHEN TASK IS NEVER TO BE CHECKPOINTED
SF.SR==100000 ;++021 [15] SET WHEN TASK ALLOWS VSDR$ DIRECTIVE FROM ALL USERS
;
S.SIZ==32. ;SIZE OF STD IN BYTES
```

B.3 Active Task List (ATL)

The ATL is a priority ordered list of ATL nodes for active tasks that have memory allocated for their execution. Either the tasks represented by entries in the ATL are either memory-resident or a request for their loading has been queued. The listhead for this deque is in the system communications area (SCOM). The entries have the following format:

```
.SBTTL ATL -- ACTIVE TASK LIST
;
; ATL -- ACTIVE TASK LIST
;
; THE "ATL" IS A PRIORITY ORDERED DEQUE OF "ATL" NODES FOR ACTIVE TASKS
; THAT HAVE MEMORY ALLOCATED FOR THEIR EXECUTION.  THE TASKS REPRESENTED
; BY ENTRIES IN THE ATL ARE EITHER MEMORY RESIDENT, OR A REQUEST FOR THEIR
; LOADING HAS BEEN QUEUED.  THE LISTHEAD FOR THIS DEQUE IS IN THE SYSTEM
; COMMUNICATIONS AREA (SCOM), AND THE NODES ARE OF THE FOLLOWING FORMAT:
;
  ; WD. 00 (B 00) -- FORWARD LINKAGE
  ; WD. 01 (B 02) -- BACKWARD LINKAGE
  ; WD. 02 (B 04) -- NODE ACCOUNTING WORD (STD ENTRY ADR OF REQUESTOR)
A.RQ==N.AW
A.TI==N.TI;WD. 03 (B 06) -- TI IDENTIFICATION - PUD ADDRESS
A.RP==10 ; WD. 04 (B 10) -- TASK'S RUN PRIORITY (BYTE)
A.IR==11 ;        (B 11) -- TASK I/O IN PROCESS COUNT (BYTE)
A.IN==12 ; WD. 05 (B 12) -- TASK I/O PENDING COUNT (BYTE)
A.CS==13 ;        (B 13) -- SAVED STATUS OF CHECKPOINTED TASK
A.MT==14 ; WD. 06 (B 14) -- TASK MARK TIME PENDING COUNT (BYTE)
A.CP==15 ;        (B 15) -- SAVED PRIORITY OF CHECKPOINTED TASK (BYTE)
A.HA==16 ; WD. 07 (B 16) -- 1/64TH REAL ADR OF LOAD IMAGE
A.TS==N.SB;WD. 10 (B 20) -- TASK STATUS (BYTE)
A.AS==21 ;        (B 21) -- AST INDICATOR (PREVIOUS STATUS) BYTE
A.TD==22 ; WD. 11 (B 22) -- SYSTEM TASK DIRECTORY (STD) ENTRY ADDRESS
A.EF==24 ; WD. 12 (B 24) -- TASK'S EVENT FLAGS (1-32)
  ; WD. 13 (B 26) -- (SECOND HALF OF TASK'S EVENT FLAGS)
A.FM==30 ; WD. 14 (B 30) -- TASK'S EVENT FLAGS MASKS (64-BITS)
  ; WD. 15 (B 32) -- (SECOND WORD OF FLAGS MASK)
  ; WD. 16 (B 34) -- (THIRD WORD OF FLAGS MASK)
  ; WD. 17 (B 36) -- (FOURTH WORD OF FLAGS MASK)
  ;
  ; THE EVENT FLAG MASKS AT A.FM ARE USED FOR VARIOUS PURPOSES
  ; BY THE EXEC, TO RECORD INFORMATION ABOUT THE STATE OF A TASK.
  ; THE SIGNIFICANCE OF THESE WORDS DEPENDS ON THE STATE OF THE
  ; TASK:
  ;
  ; 1. UP TO FIRST TIME LOAD (STATES LRP, LRQ, LRS)
  ;
  ; A.FM+0 PUD ADDRESS OF DEVICE TO LOAD TASK FROM
  ; A.FM+2 ADDRESS OF STL NODE FOR THIS TASK, OR ZERO
  ; A.FM+4 UIC FOR TASK TO RUN, OR 0 IF NOT SPECIFIED
  ; A.FM+6 ATL ADDRESS OF TASK WHICH REQUESTED THIS ONE,
  ;  IF REQUESTED BY EXEC$ OR FIX$
  ;
  ; 2. WAITING OR STOPPED FOR SINGLE GROUP OF EVENT FLAGS
  ;    (STATES WF0, WF1, WF2, WF3, ST0, ST1, ST2, ST3)
  ;
  ; A.FM+0 MASK FOR FLAGS BEING WAITED FOR IN RELEVANT
  ;  EVENT FLAG WORD (A.EF+0, A.EF+2, .COMEF, .COMEF+2)
  ;
  ; 3. WAITING OR STOPPED FOR ALL GROUPS OF EVENT FLAGS (STATES WF4, ST4)
  ;
  ; A.FM+0 MASK FOR FLAGS 1-16  (A.EF+0)
  ; A.FM+2 MASK FOR FLAGS 17-32 (A.EF+2)
```

```
  ; A.FM+4 MASK FOR FLAGS 33-48 (.COMEF)
  ; A.FM+6 MASK FOR FLAGS 49-64 (.COMEF+2)
  ;
  ; 4. WAITING FOR EXECUTIVE SEMAPHORE (STATE WSM)
  ;
  ; A.FM+0 MASK FOR SEMAPHORE BEING WAITED FOR
  ;
  ; 5. AFTER TASK EXIT (STATES EXT, STN)
  ;
  ; A.FM+0 REASON FOR EXIT (LO BYTE), EXIT FLAGS (HI BYTE):
  ;
  ;  BIT 8 (000400)  SET IF TKTN REQUIRED
  ;  BIT 9 (001000) SET IF I/O RUNDOWN REQUIRED
  ;  BIT 10 (002000) SET IF TASK EXITED WITH VALID STATUS
  ;
  ; A.FM+2 TASK EXIT STATUS
  ;
  ; 6. WAITING FOR DIRECTIVE (STATE WDI)
  ;
  ;    MEANING DEPENDS ON PARTICULAR DIRECTIVE.  CURRENTLY THIS IS
  ;    USED FOR:
  ;
  ; EXEC$, FIX$:
  ;
  ;    A.FM+6 PRESET TO -03, ERROR CODE FOR 'INSUFFICIENT
  ;   MEMORY'
  ;
  ;
  ; 7. DIRECTIVE FAILED (STATE DIF)
  ;
  ;    MUST BE SET BY CODE WHICH PUTS TASK IN THIS STATE TO:
  ;
  ; A.FM+6 ERROR CODE TO RETURN TO TASK'S DSW
  ;
  ;  NOTE THAT A.FM+0 CANNOT BE USED BECAUSE THIS STATE
  ;  OCCURS FOR A TASK AFTER IT HAS EXITED, WHEN
  ;  .TKTN. IS REQUESTED
  ;
A.PD==40 ; WD. 20 (B 40) -- TASK'S RUN PARTITION (TPD ADDRESS)
A.AF==42 ; WD. 21 (B 42) -- AST DEQUE LISTHEAD (FWD POINTER)
A.AB==44 ; WD. 22 (B 44) -- AST DEQUE LISTHEAD (BKWD POINTER)
A.SA==46 ; WD. 23 (B 46) -- SWAP ADDRESS
A.TZ==50 ; WD. 24 (B 50) -- CURRENT TASK SIZE  ++023
A.TF==52 ; WD. 25 (B 52 -- TASK FLAGS
A.SD==54 ; WD.26 (B. 54) -- ALLOCATION FACTOR
  ;     DISC ADDRESS IN A.IA
A.QI==55 ;        (B. 55) -- COUNT OF ACTIVE AND QUEUED I-O
A.SW==56 ; WD.27 (B. 56) -- COUNT OF SWAP I-O
A.SS==57 ; (B. 57) -- SAVE STATUS FOR IAS SUSPEND
;
; TASK STATUS VALUES ARE DESCRIBED AT 'ASXDT'
;
;
AF.CP==001 ; SET WHEN TASK IS CHECKPOINTED
AF.SA==002 ; Set when task was running in super mode prior to AST
AF.AD==004 ; SET WHEN TASK AST RECOGNITION IS INHIBITED
AF.CD==010 ; SET WHEN CHECKPOINTING IS DISABLED
AF.MC==020 ; SET WHEN TASK IS MARKED FOR CHECKPOINTING
AF.KA==040 ; SET WHEN TASK HAS A KERNAL AST QUEUED+
AF.IO==100 ; SET WHEN TASK HAS AN I/O COMPLETION EVENT IN ITS AST QUEUE
AF.PF==200 ; SET WHEN THERE IS A POTENTIAL POWER FAIL AST  ; +++010
AF.RR==400 ; SET WHEN POTENTIAL RECEIVE BY REFERENCE AST
AF.BF==1000 ; SET WHEN A TASK IS TO BE FIXED
```

```
AF.FX==2000 ; SET WHEN A TASK IS FIXED
AF.AS==4000 ; SET WHEN AN AST HAS BEEN DECLARED
AF.RA==10000 ; SET WHEN THERE IS A POTENTIAL RECEIVE AST
AF.RL==20000 ; SET IF TASK NEEDS TO BE RELOADED
AF.IA==40000 ; SET IF THE TASK IS IAS CONTROLLED
AF.TR==100000 ; SET IF THE TASK IS DOING TT READ
;
A.SIZ==48. ;SIZE OF ATL IN BYTES
;
;
; IF A TIMESHARING TASK THE ATL WILL BE 8 WORDS LARGER AN CONTAIN
; THE FOLLOWING ADDITIONAL INFORMATION
;
A.TUF==60; WD. 30 (B 60) -- UTL FORWARD POINTER
A.TUB==62; WD. 31 (B 62) -- UTL BACKWARD POINTER
A.TFW==64; WD. 32 (B 64) -- TIMESHARING FLAGS WORD
A.TST==66; WD. 33 (B 66) -- TIMESHARING STATUS BYTE
A.TSV==67;    (B 67) -- STATUS SAVE
A.JN==70 ; WD. 34 (B 70) -- JOB NODE ADDRESS  ++023
A.TAI==72; WD. 35 (B 72) -- ACCOUNTING STATE   ++032
  ; VALUE: 0 - TASK LOADING     (NO INFO)   ++032
  ;  2 - TASK SWAPPED OUT (INFO IN UJN) ++032
  ;  4 - TASK IN MEMORY   (INFO IN HEADER) ++032
  ;  6 - TASK EXITING     (INFO IN ATL E.TAC)++032
  ;    (B 73) -- (SPARE)    ++032
A.TQU==74; WD. 36 (B 74) -- QUANTUM
A.TLV==76; WD. 37 (B 76) -- UTL LEVEL LIST HEAD
;
A.TSIZ==64.
;
; TIMESHARING TASK FLAGS WORD BIT DEFINITIONS
;
AT.NL == 001 ;FIRST TIME LOAD
AT.TR == 002 ;SET IF TASK IS RESIDENT
AT.TL == 004 ;SET IF TASK IS TO BE LOADED
AT.IA == 010 ;SET IF INSTALL IS ACTIVE (OR TO BE RUN)
;  ;++018 UNUSED
;  ;++018 UNUSED
AT.IB == 100 ; SET IF TASK TO BE ABORTED WHEN INSTALL IS COMPLETE
AT.LS == 200 ; SET IF LUNS NEED TO BE REASSIGNED
AT.DS == 400 ; DELETE STD NODE ON EXIT
AT.TH == 1000 ; TEMPORARY HIGH-PRIORITY, USED TO FORCE LOADING ;++015
AT.BT == 2000 ; BATCH TASK (CLI OR USER TASK)
AT.SA == 4000 ; TASK NON-SWAPPABLE FOR ABORT
AT.TA == 10000 ; TASK IS ON THE ATL OR IS BEING INSTALLED
AT.LD == 20000 ; TASK IS LOADING
AT.DB == 40000 ; TASK IS TO BE INTERRUPTED FOR DEBUGGING AID
AT.HP == 100000 ; TASK IS TO BE RUN AT HIGH ATL PRIORITY     ;++015
;
;
;
;
; TIMESHARING TASK STATUS BYTE VALUES
;
; THESE VALUES ARE USED IN A.TST TO CONTROL TASK OPERATION WITH RESPECT
; TO THE TIMESHARING SCHEDULER (TSSHED).
;
JS.RUN ==00  ; TASK RUNNABLE
;
JS.RSD ==02  ; TASK TO BE SUSPENDED
;
JS.SUS ==04  ; TASK IS SUSPENDED
;
```

```
JS.ABT ==06  ; TASK TO BE ABORTED
;
JS.NEW ==10  ; TASK NEW TO SCHEDULER
;
JS.EXT ==12  ; TASK EXITED (BUT NOT YET PROCESSED BY TCP)
;
JS.LOD ==14  ; TASK TO BE LOADED
;
JS.CON ==16  ; TASK TO BE CONTINUED
;
JS.NW2 ==20  ; TASK NEW AFTER INSTALL
;
JS.EXX ==22  ; TASK EXITING, TCP QIO PENDING
;
JS.FIN ==24  ;++026 TASK EXITED AND PROCESSED BY TCP (IE UJN RELEASED)

;
; TIMESHARING ATL LINKAGE
;
; FOR TIMESHARING TASKS THE ATL IS ALSO LINKED INTO LEVELS ACCORDING
; TO THE PREVIOUS ACTIVITY OF THE TASK.  MOST SERVICING OF TIMESHARING
; TASK'S ATLS IS DONE WITH A REGISTER ADDRESSING THE UTL POINTER.
; THE FOLLOWING OFFSETS ARE DEFINED SO THAT THE WHOLE ATL CAN BE
; REFERENCED WHEN A REGISTER POINTS TO THE UTL (A.TUF)
;
;
X.RQ==A.RQ-A.TUF
X.TI==A.TI-A.TUF
X.RP==A.RP-A.TUF
X.IR==A.IR-A.TUF
X.IN==A.IN-A.TUF
X.CS==A.CS-A.TUF
X.MT==A.MT-A.TUF
X.CP==A.CP-A.TUF
X.HA==A.HA-A.TUF
X.NA==A.HA-A.TUF
X.TS==A.TS-A.TUF
X.AS==A.AS-A.TUF
X.TD==A.TD-A.TUF
X.EF==A.EF-A.TUF
X.FM==A.FM-A.TUF
X.PD==A.PD-A.TUF
X.AF==A.AF-A.TUF
X.AB==A.AB-A.TUF
X.SA==A.SA-A.TUF
X.TZ==A.TZ-A.TUF     ; ++023
X.TF==A.TF-A.TUF
X.SD==A.SD-A.TUF
X.QI==A.QI-A.TUF
X.SW==A.SW-A.TUF
X.SS==A.SS-A.TUF
X.UF==A.TUF-A.TUF
X.UB==A.TUB-A.TUF
X.FW==A.TFW-A.TUF
X.ST==A.TST-A.TUF
X.SV==A.TSV-A.TUF
X.JN==A.JN-A.TUF     ; ++023
X.AI==A.TAI-A.TUF     ; ++032
X.QU==A.TQU-A.TUF
X.LV==A.TLV-A.TUF
```

B.4 I/O Request Node

I/O request nodes (i.e., queue elements) have the following format:

```
.SBTTL IRQ -- I/O REQUEST QUEUE
;
; IRQ -- I/O REQUEST QUEUE
;
; THE "IRQ" IS A PRIORITY ORDERED DEQUE OF I/O REQUEST NODES WITH ITS
; LISTHEAD IN THE PUD ENTRY OF THE PHYSICAL UNIT FOR WHICH THE I/O
; REQUEST WAS QUEUED.  EACH PHYSICAL UNIT HAS ITS OWN I/O REQUEST QUEUE.
; I/O REQUEST NODES ARE CREATED AND QUEUED PRIMARILY BY THE "QUEUE I/O"
; DIRECTIVE.  HOWEVER, THE EXEC ALSO CREATES I/O REQUESTS TO:
; (1) LOAD A TASK IMAGE, (2) RECORD A TASK IMAGE [CHECKPOINTING], AND
; (3) TO RUNDOWN I/O ON AN EXIT'ED TASK.  I/O REQUEST NODES ARE OF
; THE FOLLOWING FORMAT.
;
  ; WD. 00 (B 00) -- FORWARD LINKAGE
  ; WD. 01 (B 02) -- BACKWARD LINKAGE
  ; WD. 02 (B 04) -- NODE ACCOUNTING WORD (STD ENTRY ADR OF REQUESTOR)
R.TD==N.AW
R.AT==06 ; WD. 03 (B 06) -- ATL NODE OF REQUESTOR ***
R.PR==10 ; WD. 04 (B 10) -- PRIORITY (BYTE)
R.DP==11 ;        (B 11) -- DPB SIZE (BYTE) ***
R.LU==12 ; WD. 05 (B 12) -- LOGICAL UNIT NUMBER (BYTE)
R.FN==13 ;        (B 13) -- EVENT FLAG NUMBER (BYTE)
R.FC==14 ; WD. 06 (B 14) -- I/O FUNCTION CODE
R.SB==16 ; WD. 07 (B 16) -- VIRTUAL ADDRESS OF STATUS BLOCK
R.AE==20 ; WD. 10 (B 20) -- VIRTUAL ADDRESS OF AST SERVICE ENTRY
R.UI==22 ; WD. 11 (B 22) -- USER IDENTIFICATION CODE
R.PC==22 ;        (B 22) -- PROGRAMMER CODE
R.GC==23 ;        (B 23) -- GROUP CODE
R.PB==24 ; WD. 12 (B 24) -- PARAMETER #1
  ; WD. 13 (B 26) -- PARAMETER #2
  ; WD. 14 (B 30) -- PARAMETER #3
  ; WD. 15 (B 32) -- PARAMETER #4
  ; WD. 16 (B 34) -- PARAMETER #5
  ; WD. 17 (B 36) -- PARAMETER #6
R.PD==40 ; WD. 20 (B 40) -- PUD POINTER FOR THIS REQUEST
R.EL==42 ; WD. 21 (B 42) -- ERROR LOG BUFFER POINTER/FLAG
R.WA==44 ; WD. 22 (B 44) -- FLAG BYTE FOR EXEC
R.HF==45 ; WD. 22 (B 45) -- WORK AREA FOR DEVICE HANDLERS (Handler Flags)
  ; WD. 23 (B 46) -- WORK AREA FOR DEVICE HANDLERS
  ; WD. 24 (B 50) -- WORK AREA FOR DEVICE HANDLERS
R.IA==52 ; WD. 25 (B 52) -- ASR3 VALUE FOR BUFFER BASE(=-1 FOR SCOMM)
R.IB==54 ; WD. 26 (B 54) -- EITHER:
  ;
  ;    1. INTERMEDIATE BUFFER ADDRESS (RSX
  ;       INTERMEDIATE BUFFERED DEVICES)
  ;    2. TPD ADDRESS FOR PARTITION (IAS
  ;       EXEC LOAD/RECORD REQUESTS)
  ;    3. ADDRESS OF BLOCK LOCK NODE (FILE
  ;       STRUCTURED DEVICES)
  ;
R.UB==56 ; WD. 27 (B 56) -- EITHER:
  ;
  ;    1. USER BUFFER ADDRESS (RSX INTERMEDIATE
  ;       BUFFERED DEVICES)
  ;    2. MUL NODE ADDRESS (IAS)
;
; FOR EXECUTIVE I/O REQUESTS A LARGER NODE IS USED TO ALOOW
; TRANSFERS GREATER THAN 32K-32 WORDS
;
```

```
R.BA==60 ; WD. 30 (B 60) -- BASE ADDRESS OF COMPLETE TRANSFER (MOD 64)
R.TB==62 ; WD. 31 (B 62) -- BASE ADDRESS CURRENT TRANSFER (MOD 64)
R.TS==64 ; WD. 32 (B 64) -- TRANSFER SIZE (MOD 64)
R.BN==66 ; WD. 33 (B 66) -- CURRENT BLOCK NUMBER (HI)
  ; WD. 34 (B 70) -- CURRENT BLOCK NUMBER (LO)
;
RS.BLK==127.  ; MAXIMUM NUMBER OF 256. WORD DISC BLOCKS IF
   ; NOT LAST TRANSFER
;
RS.32W==RS.BLK*10 ; MAXIMUM TRANSFER SIZE IN 32. WORD BLOCKS IF
   ; NOT LAST TRANSFER
RS.MAX==RS.32W+7 ; MAXIMUM TRANSFER SIZE IN 32. WORD BLOCKS IF
   ; LAST TRANSFER
;
; THE LOW ORDER THREE-BITS OF THE I/O FUNCTION CODE ARE USED BY THE SYSTEM
; AS FOLLOWS:
;
RF.IT==000001 ;[0] -- RESERVED FOR FUTURE USE
RF.XR==000002 ;[1] -- "EXPRESS REQUEST"
RF.IR==000004 ;[2] -- RESERVED FOR FUTURE USE
RF.GC==000040 ;[5] -- GCD RECORD REQU. NODE INDICATOR
; IAS EXECUTIVE I-O FLAGS
;
RW.LK==200 ; SET IF MEMORY LOCKED FOR REQUEST (MUL ADDRESS IN R.UB)
RW.ML==100 ; SET IF NODE (GCD OR ATL) ADDRESS STORED IN R.UB
RW.IA==010 ; SET IF AN IAS SWAP REQUEST
RW.SW==020 ; SET IF THE SWAP COUNT IS INCREMENTED FOR REQUEST
RW.SP==004 ; SET IF REQUEST IS TO OUTPUT SPOOLED DEVICE ;++017/16
;
R.SIZ == 60 ; SIZE OF TASK REQUEST NODE IN BYTES
;
R.XSIZ== 100 ; SIZE OF EXECUTIVE REQUEST NODE IN BYTES
;
; Flags used with the internal handlers' work area (R.HF)
;
RHF.AB== 1    ; Handler's per request aborted bit
RHF.RN== 2    ; Release request node address for error log
RHF.MS== 4    ; Multicopy Structure function in progress
RHF.EL== 10    ; BBR request to log error (ER$LOG)          050
RHF.BB== 200    ; Request owned by HIBBR task

;
; *** WHENEVER AN I/O REQUEST IS QUEUED BY THE "QUEUE I/O" DIRECTIVE, THE
; DPB SIZE AND THE REQUESTOR'S ATL NODE ADDRESS ARE RECORDED IN THE I/O
; REQUEST NODE.  WHENEVER AN I/O REQUEST IS QUEUED AS A RESULT OF ANOTHER
; DIRECTIVE (VIZ., "REQUEST" CAUSING A TASK IMAGE TO BE LOADED), THE DPB
; SIZE AND THE REQUESTOR'S ATL NODE ADDRESS ARE SET TO ZERO.  THUS, BOTH
; BOTH THE DPB SIZE AND THE ATL NODE ADDRESS ARE ALSO "EXEC REQUEST"
; INDICATORS.
```

B.5 Interrupt Service Routine Node

ISR nodes have the following format:

```
;    ;WD. 00 -- FORWARD POINTER (UNUSED)
;    ;WD. 01 -- BACKWARD POINTER (UNUSED)
;    ;WD. 02 -- STD NODE OF HANDLERS TASK (USED FOR ACCOUNTING)
;    ;WD. 03 -- INTERRUPT SERVICE ENTRY POINT -- 1ST WD OF MOV
;    ;WD. 04 -- 2ND WORD OF MOV @#KP.AR3,-(SP)
;    ;WD. 05 -- 1ST WORD OF BIC #0,@#0 INSTR
;    ;WD. 06 -- 2ND WORD OF BIC #0,@#0 INSTR
;    ;WD. 07 -- 3RD WORD OF BIC #0,@#0 INSTR
;    ;WD. 10 -- 1ST WORD OF MOV @#IPAR3,@#KP.AR3
;    ;WD. 11 -- 2ND WORD OF MOV @#IPAR3,@#KP.AR3
;    ;WD. 12 -- 3RD WORD OF MOV @#IPAR3,@#KP.AR3
;NOTE:     ALL CONDITION BITS MMUST BE CLEARED AFTER THIS INSTRUCTION
;    ;WD. 13 -- JMP INTERRUPT SERVICE ROUTINE OR SET CONDITION CODES
;    ;WD. 14 -- 2ND WORD OF JUMP (NO CC'S SET) OR 1ST WORD JMP
;    ;WD. 15 -- 2ND WORD OF JUMP (CC'S SET)
;    ;WD. 16 -- 17 -- UNUSED
```

C I/O Status Block

Figure C–1 illustrates the normal usage of the I/O status block:

Figure C–1 I/O Status Block

	high byte	low byte
Word 0	additional status information	success/failure status indicator
Word 1	number of bytes transmitted	

The low byte of word zero should be set to one of the system defined I/O status values (symbols of the form IS.xxx or IE.xxx). Use IS.SUC to indicate that the request has been completed successfully. Select other success and failure codes appropriately for the request status; for example, use IE.DNR if a device is not ready (for instance, because a disk drive was not loaded with a disk).

The high byte of word zero is normally set to zero. However, it can be used to return additional information to qualify the status in the low byte. For example, the terminal handler uses this byte to return the character which successfully terminated a read.

The second word of the status block is used during transfer requests (read or write) to contain the number of bytes actually transferred. This information is used by many system tasks and must always be set up correctly. For non-transfer requests, this word can be used in any way. For example, it is used by the terminal handler in response to the get single characteristic QIO to return the current setting of the corresponding characteristic.

D Sample Device Handlers

Sources for device handlers are distributed with every system. The distributed handler that is closest in function to the handler to be written should be assembled and listed. The following are likely to be of particular interest:

1 [311,14]PR.MAC - paper tape reader. Shows a simple non-multi-user handler for a non-DMA device.

2 [311,14]LP.MAC - line printer handler. Shows a simple multi-user handler.

3 [311,14]DK.MAC - RK05 handler. Shows a simple handler for a DMA device.

4 [311,14]DM.MAC - RK06 handler. Shows a complex disk handler, including overlapped and optimized seeks.

Index

A

B

C

D

E

F

H

I

T

U

V

W

Reader's Comments

This form is for document comments only. Digital will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)_____

Name_____Date_____

Organization_____

Street_____

City_____State_____Zip Code_____
or Country

Do Not Tear - Fold Here and Tape

digital™

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 33 MAYNARD MASS

POSTAGE WILL BE PAID BY ADDRESSEE

IAS Engineering/Documentation
Digital Equipment Corporation
5 Wentworth Drive GSF/L20
Hudson, NH 03051-4929

Do Not Tear - Fold Here